Herbert Schildt

C++ manual complet

Traducerea
Mihai Dăbuleanu

Teora

Titlul original: C++ THE COMPLETE REFERENCE
Traducere după ediția originală în limba engleză, publicată de Osborne McGraw-Hill.

Prima ediție: 1997
Retipărită: aprilie 1998

Distribuție

București: B-dul Al. I. Cuza nr. 39; tel./fax: 222.45.33
Sibiu: Șos. Alba Iulia nr. 40; tel.: 069/21.04.72; fax: 069/23.51.27
Bacău: Calea Mărășești nr. 5; tel./fax: 034/18.18.26

Teora – Cartea prin poștă

CP 79-30, cod 72450 București, România
Tel./Fax: 252.14.31

Teora

CP 79-30, cod 72450 București, România
Fax: 210.38.28

NOT 2394 CAL C++, MANUAL COMPLET
ISBN 973-601-595-5
Printed in Romania

CUPRINS

Partea a III-a
Câteva aplicaţii de C++

Despre autor

Herbert Schildt este cel mai cunoscut autor de manuale și programe de C/C++ din întreaga lume. Cărțile lui de programare s-au vândut în peste un milion și jumătate de exemplare în întreaga lume și au fost traduse în toate limbile de circulație mondială. El este autorul cărților de succes **C: The Complete Reference (C: Manual Complet)**, acum la cea de a treia ediție, **Teach Yourself C (Învață singur C)** și **Teach Yourself C++ (Învață singur C++)**. A mai scris, de asemenea, **The Annotated ANSI C Standard, C++ from the Ground Up, Schildt's Windows 95 Programming in C and C++** și numeroase alte cărți. Schildt este președintele companiei Universal Computing Laboratories, o firmă de consultanță software din Mahomet, Illinois și membru al comitetului de standardizare ANSI C++. El deține titlul de Master în informatică al Universității Illinois.

Prefață

Noutatea cu adevărat importantă în această carte este aceea că a început lucrul la ANSI Standard C++. Chiar dacă procesul de standardizare este unul lent și va mai dura probabil mulți ani, crearea unui standard pentru C++ este pasul final necesar stabilirii acestuia ca limbaj de programare de clasă profesională mondială. Desigur, informațiile conținute aici reflectă varianta de C++ propusă pentru standardizarea ANSI.

Chiar dacă C++ face parte din universul programării deja de mai mulți ani, este bine să ne reamintim că el încă reprezintă un pas mare în programare. Bazat pe limbajul C, C++ adaugă extinderi care admit programarea orientată pe obiecte. Aceste extinderi au mărit enorm, în general, puterea limbajului. Limbajele de programare (și metodologiile subordonate) s-au dezvoltat constant de la inventarea lor în anii '50. C++ este pasul următor. Pe lângă descrierea programării orientate pe obiecte, pe care o conține în cadrul său, scopul principal al cărții este ca dvs., ca programator, să puteți lucra cu programe din ce în ce mai mari și mai complexe. În această privință C++ reușește admirabil.

C++ este construit pe structura limbajului C. Acum, în momentul în care scriu această carte, C este încă cel mai popular şi mai important limbaj de programare din lume. Deoarece reprezintă o dezvoltare şi o extindere a sa, se aşteaptă ca C++ să-şi lărgească în continuare sfera acceptării şi utilizării. Forţa şi flexibilitatea lui, combinate cu faptul că se bazează pe popularitatea limbajului C, îi asigură deja locul în istoria programării.

O carte pentru toţi programatorii

Această carte cuprinde atât caracteristicile de tip C ale limbajului C++, cât şi aspectele sale specifice. Totuşi, cea mai mare atenţie se acordă acelor caracteristici care sunt proprii limbajului C++. Deoarece mulţi cititori sunt deja familiarizaţi şi competenţi în C, elementele preluate din C sunt prezentate separat de cele specifice limbajului C++. Acest fapt evită ca programatorii cunoscători de C să se „bălăcească" prin maldăre de informaţii pe care le cunoaşte deja, ei putând trece direct la secţiunile din carte care acoperă caracteristicile specifice limbajului C++.

Acest manual de referinţă pentru C++ este destinat tuturor programatorilor în C++, indiferent de nivelul lor de experienţă. El presupune, însă, un cititor capabil să creeze măcar un program simplu. Dacă de-abia acum învăţaţi C++, această carte vă va fi o companie excelentă pentru orice program explicativ C++ şi va servi ca sursă de răspunsuri la întrebările dvs.

De aceea, fie că sunteţi un programator cu experienţă în C care învaţă C++ fie un novice în programare, veţi găsi această carte ca fiind de mare utilitate.

Conţinutul

Această carte descrie în detaliu toate caracteristicile limbajului C++, inclusiv fundamentul său: C. Cartea este împărţită în trei părţi, care cuprind:

- Bazele limbajului C
- Limbajul C++
- Exemple de aplicaţii C++

Partea întâi oferă o prezentare cuprinzătoare a bazei limbajului C++: limbajul C. Aici este descris în întregime standardul ANSI C. Cunoaşterea în profunzime a limbajului C este o premisă obligatorie pentru învăţarea limbajului C++. Partea a doua prezintă în detaliu extinderile şi îmbunătăţirile adăugate limbajului C de către C++. Partea a treia oferă exemple practice de aplicaţii de C++ şi de programare orientată pe obiecte.

Partea I

Baza limbajului C++: Limbajul C

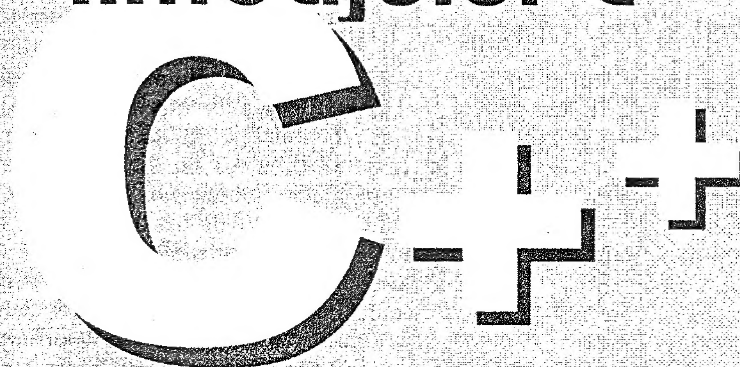
Partea întâi a acestei cărţi prezintă caracteristicile de tip C ale limbajului C++. După cum probabil ştiţi, C++ este construit pe baza limbajului C. C++ a fost inventat pornind de la C. Acestuia i-au fost adăugate noi facilităţi şi extinderi destinate acceptării programării orientate pe obiecte (OOP). Dar aspectele de tip C ale limbajului C++ nu au fost niciodată abandonate.

În forma obişnuită, C++ este o versiune dezvoltată a limbajului C standard ANSI, care este, de fapt, un *document de bază* pentru standardul ANSI C++ propus. Din acest motiv, orice compilator de C++ este, prin definiţie, şi un compilator de C. Deoarece C++ este construit pe baza limbajului C, nu puteţi programa în C++ fără să ştiţi să programaţi în C. Mai mult, mare parte dintre conceptele fundamentale din C sunt astfel şi pentru C++.

Atât timp cât C++ este un super-C, cele descrise în prima parte a cărţii sunt pe deplin aplicabile şi pentru C++. Caracteristicile proprii acestuia sunt detaliate în Partea a Doua. Motivul pentru care aspectele de tip C ale limbajului C++ sunt prezentate într-o secţiune proprie este intenţia de a face mai uşor programatorilor cu experienţă în C accesul rapid la informaţii despre C++ fără să se „bălăcească" prin maldăre de informaţii pe care le cunosc deja. Pe parcursul întregii Părţi întâi, sunt evidenţiate diferenţele minore între C şi C++.

Capitolul 1

O privire de ansamblu asupra limbajului C



Scopul acestui capitol este să prezinte o vedere de ansamblu a limbajului de programare C, a originilor, utilizărilor şi a filosofiei care stă la temelia lui. Deoarece C++ este construit pe baza limbajului C, acest capitol oferă o importantă perspectivă istorică asupra originilor sale.

Originile limbajului C++

C a fost inventat şi implementat prima oară de Dennis Ritchie pe un DEC PDP-11 care utiliza sistemul de operare UNIX. C este rezultatul unui proces de dezvoltare care a început cu un limbaj numit BCPL, creat de Martin Richards; BCPL a influenţat un limbaj numit B, care a fost inventat de Ken Thompson. În anii '70, B a dus la dezvoltarea limbajului C.

Mulţi ani standardul de facto pentru C a fost versiunea ce însoţea sistemul de operare UNIX. El a fost descris pentru prima dată în *The C Programming Language* (Limbajul de programare C) de Brian Kernighan şi Dennis Ritchie (Englewood Cliffs, NJ: Prentice Hall, 1978). O dată cu creşterea popularităţii calculatoarelor personale, au fost create numeroase implementări de C. A fost aproape un miracol că acestea au fost în mare măsură compatibile (însemnând că un program scris într-una din ele putea fi compilat de obicei cu succes utilizând o alta). Totuşi, deoarece nu exista nici un standard, apăreau discrepanţe. Pentru a remedia acest fapt, în vara lui 1983 a fost stabilit un comitet pentru crearea unui standard ANSI (American National Standards Institute) care să definească o dată pentru totdeauna limbajul C. Procesul de standardizare a durat şase ani (mult mai mult decât s-ar fi aşteptat orice om rezonabil). În sfârşit, standardul ANSI C a fost adoptat în decembrie 1989, primele copii devenind disponibile la începutul lui 1990. Astăzi, toate compilatoarele C/C++ se supun standardului ANSI C. De asemenea, standardul ANSI C este o bază pentru propunerea de standard ANSI C++.

C este un limbaj de nivel mediu

C este deseori numit un limbaj *de nivel mediu*. Aceasta nu înseamnă că C este mai puţin performant, mai greu de utilizat sau mai puţin dezvoltat decât un limbaj de nivel înalt, cum ar fi BASIC sau Pascal, şi nici că are natura greoaie a unui limbaj de asamblare (inclusiv problemele sale). C este descris ca limbaj de nivel mediu deoarece combină cele mai bune facilităţi ale unui limbaj de nivel înalt cu posibilităţile de control şi flexibilitatea limbajului de asamblare. **Tabelul 1-1** prezintă locul limbajului C în gama limbajelor informatice.

Ca limbaj de nivel mediu, C permite lucrul cu biţi, octeţi şi adrese - elementele de bază cu care funcţionează calculatorul. În ciuda acestui fapt, codul C este, de asemenea, foarte portabil. *Portabilitatea* înseamnă că un software scris pentru un anumit tip de calculator sau sistem de operare se adaptează uşor pe oricare altul. De exemplu, un program este portabil dacă, fiind scris pentru sistemul DOS, puteţi face uşor conversia sa, astfel încât să ruleze în Windows.

Limbaje de nivel înalt	Ada Modula-2 Pascal COBOL FORTRAN BASIC
Limbaje de nivel mediu	C++ C FORTH
Limbaje de nivel scăzut	Macro-asamblor Limbaj de asamblare

Tabelul 1-1. *Locul lui C în lumea limbajelor*

Toate limbajele de programare acceptă conceptul de tip de date. Un *tip de date* defineşte o mulţime de valori pe care le poate lua o variabilă, împreună cu un ansamblu de operaţii care pot fi efectuate asupra sa. Tipurile uzuale de date sunt întreg, caracter şi real. Chiar dacă C are cinci tipuri de date de bază, el nu este un limbaj centrat pe tipurile de date, aşa cum sunt Pascal şi Ada. C permite aproape toate conversiile de tipuri. De exemplu, puteţi amesteca într-o expresie date de tip caracter cu date de tip întreg.

Spre deosebire de un limbaj de nivel înalt, C nu efectuează aproape nici un control al erorilor în timpul rulării. De exemplu, nu se face nici o verificare de încadrare în dimensiunile unei matrice. Acest tip de control cade în responsabilitatea programatorului.

În aceeaşi ordine de idei, C nu solicită o compatibilitate strictă între un parametru şi un argument. După cum probabil ştiţi din experienţa de programator, un limbaj de nivel înalt cere ca tipul unui argument să fie (mai mult sau mai puţin) identic cu cel al parametrului căruia îi va da valoarea sa. Restricţia aceasta nu este însă valabilă şi pentru C. C permite ca un argument să fie de orice tip, atât timp cât el poate fi convertit convenabil în tipul de dată al parametrului, conversie pe care dealtfel C o efectuează automat.

O caracteristică specială a limbajului C este posibilitata de a opera direct cu biţi, octeţi, cuvinte şi pointeri, ceea ce îl face foarte potrivit pentru programare la nivelul de sistem, unde toate aceste operaţii sunt foarte necesare.

Un alt aspect important al limbajului C este că el are doar 32 de cuvinte-cheie (27 din standardul inițial al lui Kernighan și Ritchie și cinci adăugate de comitetul de standardizare ANSI); ele sunt comenzile care formează limbajul C. Limbajele de nivel înalt au, în general, de câteva ori mai multe cuvinte cheie. De exemplu, cele mai multe versiuni de BASIC au cu mult peste 100 de cuvinte-cheie!.

C este un limbaj structurat

În experiența dvs. anterioară ca programator, probabil că ați auzit termenul *structură-în-blocuri* aplicat unui limbaj de calculator. Chiar dacă termenul „structurat în blocuri" nu se aplică strict limbajului C, acesta este numit în mod curent mai simplu, *limbaj* structurat. Există multe similitudini cu alte limbaje structurate, așa cum ar fi ALGOL, Pascal și Modula-2.

NOTĂ: Motivul pentru care C (ca și C++) nu este, practic, un limbaj structurat în blocuri este acela că limbajele cu această caracteristică permit să fie declarate proceduri sau funcții în interiorul altor proceduri sau funcții. Astfel, deoarece C nu permite crearea de funcții în interiorul funcțiilor, el nu poate fi numit formal structurat în blocuri.

Caracteristica distinctivă a unui limbaj structurat este *compartimentarea* codului și a datelor. Aceasta este capacitatea unui limbaj de a despărți și a ascunde de restul unui program toate informațiile și instrucțiunile necesare efectuării unei anumite sarcini. O modalitate de realizare a compartimentării este utilizarea de subrutine care folosesc variabile locale (temporare). Utilizând variabile locale, puteți scrie modulele asfel încât ceea ce se întâmplă în interiorul lor să nu aibă efecte în alte secțiuni ale programului. Având această facilitate, programelor în C li se poate împărți foarte ușor codul în secțiuni. Ca să creați funcții compartimentate, trebuie să știti doar ce face funcția, nu și cum o face. Rețineți că utilizarea excesivă a variabilelor globale (variabile cunoscute de întreg programul) permite greșelilor să se strecoare în program, prin apariția de efecte secundare nedorite. (Oricine a programat în BASIC standard cunoaște foarte bine această problemă.)

NOTĂ: Conceptul de compartimentare este dezvoltat foarte mult de extensia limbajului C, C++. Specific acestuia este faptul că o parte a programului poate să controleze foarte strict la care alte secțiuni este permis accesul.

Un program structurat vă oferă o mare varietate de posibilități de programare. El include mai multe construcții de bucle, cum ar fi **while**, **do-while** și **for**. Într-un limbaj structurat, utilizarea instrucțiunii **goto** este interzisă sau descurajată, aceasta nefiind forma obișnuită de control al programului (așa cum este, de exemplu, în BASIC standard și în clasicul FORTRAN). Un limbaj structurat vă permite să plasați instrucțiuni oriunde pe o linie și nu necesită un concept strict

despre câmp (așa cum o fac câteva din variantele mai vechi de FORTRAN).
Iată câteva exemple de limbaje structurate și nestructurate:

Nestructurate	**Structurate**
FORTRAN	Pascal
BASIC	Ada
COBOL	C++
	C
	Modula-2

Limbajele structurate tind să fie moderne. De fapt, o caracteristică a limbajelor de programare mai vechi este aceea că nu sunt structurate. Astăzi, majoritatea programatorilor consideră că în limbajele structurate este mai ușor să scrii și să întreții un program.

Componenta structurală principală din C este *funcția* - rutina de sine stătătoare a limbajului C. În C, funcțiile sunt construcțiile de blocuri în care are loc întreaga activitate a programului. Ele permit definirea și scrierea codurilor pentru sarcinile individuale ale programului, ducând la modularizarea acestora. După ce ați construit o funcție, puteți fi siguri că ea lucrează corect în diverse situații fără să inducă efecte secundare în alte părți ale programului. Crearea de funcții de sine stătătoare este foarte importantă în proiectele mari, în care codul unui programator nu trebuie să afecteze accidental un altul.

Alt mod de a structura și de a compartimenta un cod în C este utilizarea blocurilor de cod. Un *bloc de cod* este un grup de instrucțiuni alăturate logic, tratate ca un singur element. În C, creați un bloc de cod incluzând între acolade un grup de instrucțiuni. În următorul exemplu,

```
if (x < 10) {
printf("Prea jos, mai incearca o data.\n");
scanf("%d", &x);
}
```

ambele instrucțiuni de după **if** și dintre paranteze sunt ambele executate dacă **x** este mai mic decât 10. Aceste două instrucțiuni, împreună cu acoladele, reprezintă un bloc de cod. Ele constituie un singur element logic: una din instrucțiuni nu se poate executa fără ca să se execute și cealaltă. Rețineți că orice instrucțiune în C poate fi simplă sau un bloc de instrucțiuni. Blocurile de cod permit realizarea multor algoritmi cu limpezime, eleganță și eficiență. Mai mult, ele ajută programatorul să clarifice adevărata natură a algoritmului.

C este un limbaj al programatorului

Surprinzător, dar nu toate limbajele de programare sunt pentru programatori. Gândiți-vă la exemplele clasice de limbaje pentru utilizator, COBOL și BASIC.

COBOL nu a fost proiectat pentru a îndulci soarta programatorilor, pentru a mări siguranţa în exploatare a codului creat şi nici chiar pentru a creşte viteza de scriere a acestuia. COBOL a fost proiectat mai degrabă, cel puţin în parte, pentru a permite utilizatorilor să citească şi, s-ar părea, deşi este puţin probabil, să înţeleagă programul. BASIC a fost creat în principiu pentru a permite utilizatorilor să programeze un calculator pentru a rezolva probleme relativ simple.

În schimb, C (ca şi C++) a fost creat, influenţat şi testat de către adevăraţii programatori. Rezultatul final este acela că C oferă programatorului exact ceea ce îşi doreşte: restricţii puţine, motive puţine de nemulţumire, structuri în blocuri, funcţii de sine stătătoare şi un set compact de cuvinte-cheie. Utilizând C, puteţi să atingeţi eficienţa codului de asamblare combinată cu structura limbajului ALGOL sau Modula-2. Nu este de mirare că C şi C++ au ajuns cu uşurinţă cele mai răspândite limbaje printre cei mai buni profesionişti ai programării.

Faptul că puteţi utiliza C în locul limbajului de asamblare este un factor important pentru popularitatea sa în rândul programatorilor. Limbajele de asamblare folosesc reprezentarea simbolică a codului binar efectiv pe care calculatorul îl execută direct. Fiecare operaţie a limbajului de asamblare reprezintă o singură acţiune pentru calculator. Chiar dacă limbajul de asamblare oferă programatorului posibilitatea de a realiza sarcini cu flexibilitate şi eficienţă maximă, este evident dificil de operat cu el pentru dezvoltarea şi depanarea unui program. Mai mult, deoarece limbajul de asamblare este nestructurat, programul final va fi un cod-spaghetti - o masă încâlcită de sărituri, apelări şi indici. Această lipsă de structurare face programele scrise în limbaje de asamblare dificil de citit, de dezvoltat şi de întreţinut. Probabil că şi mai important este faptul că rutinele limbajelor de asamblare nu sunt portabile între echipamentele cu unităţi centrale de prelucrare (CPU) diferite.

Iniţial, C a fost folosit pentru programarea de sisteme. Un *program de sistem* formează o parte din sistemul de operare al calculatorului sau al accesoriilor sale. De exemplu, următoarele sunt numite uzual programe de sistem:

- Sisteme de operare
- Interpretoare
- Editoare
- Compilatoare
- Baze de date
- Foi de calcul

O dată cu creşterea în popularitate a limbajului C, mulţi programatori au început să îl utilizeze pentru a programa orice acţiune datorită portabilităţii şi eficienţei sale. Deoarece există compilatoare pentru orice calculator posibil, puteţi prelua un cod scris pentru un echipament, să îl compilaţi şi să îl rulaţi pe un altul cu relativ puţine modificări. Portabilitatea economiseşte atât timp cât şi bani. Compilatoarele de C produc coduri obiect foarte compacte şi rapide - de exemplu, mai compacte şi mai rapide decât majoritatea compilatoarelor COBOL.

În plus, programatorii folosesc C pentru toate tipurile de sarcini de programare deoarece le place! C oferă viteza limbajului de asamblare şi flexibilitatea limbajului FORTH, dar are puţine din restricţiile din Pascal sau din Modula-2. Fiecare programator în C poate să creeze şi să întreţină o bibliotecă de funcţii care a fost croită în stilul său propriu de programare şi care poate fi utilizată în multe programe diferite. Deoarece permite - ba mai mult, încurajează - compilarea independentă, C le oferă programatorilor posibilitatea să gestioneze uşor proiecte mari, cu un minim de redundanţă a efortului.

Forma unui program în C

Tabelul 1-2 prezintă cele 32 de cuvinte-cheie care, combinate cu sintaxa formală, alcătuiesc limbajul de programare C. 27 dintre ele au fost definite în versiunea C originală. Următoarele cinci au fost adăugate de către comitetul ANSI C: **enum**, **const**, **signed**, **void** şi **volatile**.

În plus, multe compilatoare de C (şi C++) au adăugate mai multe cuvinte-cheie care exploatează mai bine mediul de operare. De exemplu, multe compilatoare includ cuvinte-cheie pentru administrarea memoriei familiei de procesoare 8086, pentru a admite programare cu limbaje mixte şi pentru a avea acces la întreruperi. Iată o listă a câtorva cuvinte-cheie suplimentare utilizate curent:

asm	_cs	_ds	_es
_ss	cdecl	far	huge
interrupt	near	pascal	

auto	double	int	struct
break	else	long	switch
case	enum	register	typedef
char	extern	return	union
const	float	short	unsigned
continue	for	signed	void
default	goto	sizeof	volatile
do	if	static	while

Tabelul 1-2 *Cele 32 de cuvinte-cheie definite de către standardul ANSI C*

Compilatorul dvs. poate să conțină și alte extinderi care îl ajută să beneficieze de avantajele mediului său propriu.

Toate cuvintele-cheie în C sunt scrise cu literă mică. În C, literele mari diferă de cele mici: **else** este un cuvânt-cheie; **ELSE** nu este. Nu puteți folosi un cuvânt cheie în nici un alt scop într-un program în C - anume, nu ca variabilă sau ca nume de funcție.

Toate programele în C constau din una sau mai multe funcții. Singura funcție care trebuie să fie prezentă este numită **main()**, ea fiind prima funcție apelată când începe execuția unui program. Într-un cod în C bine scris, **main()** conține în esență un plan a ceea ce face programul. Planul este compus din funcții de apelare. Chiar dacă **main()** nu este un cuvânt-cheie, tratați-l ca și cum ar fi. De exemplu, nu încercați să folosiți **main()** ca nume de variabilă deoarece, probabil, veți deruta compilatorul.

Forma generală a unui program în C este prezentată în **Figura 1-1**, unde variabilele de la **f1()** la **fN()** reprezintă funcții definite de utilizator.

Biblioteca și editarea legăturilor

Din punct de vedere tehnic, puteți construi un program util și funcțional care constă doar din instrucțiunile pe care le-ați creat efectiv. Totuși, aceasta este o raritate

```
declaratii globale
returneaza-afiseaza main (lista de parametri)
{
    secventa de instructiuni
}
returneaza-afiseaza f1 (lista de parametri)
{
    secventa de instructiuni
}
returneaza-afiseaza f2 (lista de parametri)
{
    secventa de instructiuni
}
.
.
.
returneaza-afiseaza fN (lista de parametri)
{
    secventa de instructiuni
}
```

Figura 1-1 *Forma generală a unui program în C*

deoarece C nu asigură în cadrul actualei definiri a limbajului nici o metodă de efectuare a operațiilor de intrare/ieșire (I/O). Ca urmare, majoritatea programelor includ apeluri la diverse funcții conținute în *biblioteca standard.*

Toate compilatoarele de C se livrează cu o bibliotecă standard de funcții care efectuează cele mai uzuale sarcini. Standardul ANSI C specifică un set minim de funcții care trebuie conținute de bibliotecă. Compilatorul dvs. va conține probabil multe alte funcții. De exemplu, biblioteca standard nu definește nici o funcție grafică, dar compilatorul dvs. probabil că posedă câteva.



***NOTĂ:** Este important să înțelegeți că C++ admite întreaga bibliotecă a standardului ANSI C. Astfel, toate funcțiile limbajului standard C sunt valabile în programele pe care le scrieți atât în C cât și în C++. Desigur, C++ definește, de asemenea, câteva funcții de bibliotecă ce îi sunt proprii.*

Cei care au creat compilatorul au scris deja majoritatea funcțiilor de utilitate generală pe care le veți folosi. Când apelați o funcție care nu face parte din programul dvs., compilatorul își „amintește" numele ei. Apoi, editorul de legături combină codul pe care l-ați scris cu codul obiect ce l-a găsit în biblioteca standard. Acest proces este numit *linking (editare de legături)*. Unele compilatoare au propriul lor editor de legături, în timp ce altele îl folosesc pe cel standard, asigurat de sistemul de operare.

Funcțiile din bibliotecă sunt în format *realocabil*. Aceasta înseamnă că adresele din memorie pentru diverse instrucțiuni în cod mașină nu au fost definite în mod absolut - au fost păstrate doar informațiile de deplasament (offset). Atunci când programul dvs. este unit cu funcțiile din biblioteca standard, aceste adrese relative din memorie sunt folosite pentru a crea adresele utilizate efectiv. Există manuale tehnice și cărți care explică acest proces mai în detaliu. Dar, pentru a programa în C sau C++, nu aveți nevoie de mai multe amănunte privind procesul efectiv de realocare.

Multe dintre funcțiile care vă vor fi necesare atunci când veți scrie un program există în biblioteca standard. Ele se comportă ca blocuri pe care le puteți combina. Dacă scrieți o funcție pe care o veți utiliza mereu, puteți să o introduceți, de asemenea, într-o bibliotecă. Unele compilatoare vă permit să o includeți în biblioteca standard; altele vă obligă să vă creați o nouă bibliotecă. În orice caz, codul va fi acolo, așteptând să îl reutilizați.

Compilarea independentă

Multe din programele scurte sunt conținute în întregime într-un singur fișier sursă. Dar, o dată cu lungirea programului, va crește și timpul de compilare, care vă va pune răbdarea la încercare. De aceea, C permite unui program să fie cuprins în mai multe fișiere ce pot fi compilate independent. Când ați compilat toate fișierele, li se editează legăturile, împreună cu orice rutină necesară din bibliotecă, pentru a

forma codul obiect complet. Avantajul compilării independente este acela că, dacă modificaţi codul unui fişier, nu este necesar să recompilaţi întreg programul. Acesta economiseşte o cantitate mare de timp, pentru toate programele, în afară de cele mai simple. Manualul de utilizare al compilatorului C/C++ va conţine instrucţiunile de compilare a fişierelor multiple.

Utilizarea unui compilator de C++ pentru a compila programe în C

Programele din Partea întâi a acestei cărţi sunt programe în C. Totuşi, probabil că pentru a le compila utilizaţi un compilator de C++. Toate compilatoarele de C++ sunt şi compilatoare de C, aşa încât nu veţi avea probleme. Însă, atunci când compilaţi programe în C trebuie să reţineţi un lucru important: fişierele trebuie să aibă extensia .C (nu .CPP). În momentul scrierii acestei cărţi, majoritatea compilatoarelor C++ comercializate compilează automat fişierele cu extensia .C ca programe în C, iar fişierele cu extensia .CPP, ca programe în C++. (Unele compilatoare pot să utilizeze o conversie puţin diferită, aşa încât verificaţi în manualul dvs.) Chiar dacă C este conţinut în C++, există câteva diferenţe minore între cele două limbaje. Din acest motiv, *trebuie* să compilaţi programele în C ca *programe în C*, iar programele în C++, ca *programe în C++*.

Capitolul 2

Expresii



Acest capitol prezintă elementele fundamentale ale limbajului C (şi C++): expresiile. Aşa cum veţi vedea, expresiile în C sunt mult mai generale şi mai puternice decât în alte limbaje de calcul. Expresiile sunt formate din elementele atom ale limbajului C: *date* şi *operatori*. Datele pot fi reprezentate prin variabile sau prin constante. Ca şi majoritatea celorlalte limbaje, C acceptă mai multe tipuri de date. De asemenea, el asigură o mare varietate de operatori.

Cele cinci tipuri de date de bază

În C există cinci tipuri de date de bază: caracter, întreg, în virgulă mobilă, în virgulă mobilă cu dublă precizie şi fără nici o valoare (**char, int, float, double** şi respectiv **void**). După cum veţi vedea, toate celelalte tipuri de date din C se bazează pe acestea cinci. Dimensiunea şi domeniul de cuprindere a acestor tipuri de date pot să varieze în funcţie de tipul procesorului şi de modul de implementare a limbajului C. Dar, în toate cazurile, un caracter este reprezentat de un octet. Chiar dacă de multe ori un întreg ocupă doi octeţi, nu puteţi să vă luaţi această răspundere dacă doriţi ca programul dvs. să fie portabil în cele mai generale medii. Este important să înţelegeţi că standardul ANSI C stipulează doar domeniul de cuprindere minimal al fiecărui tip de date, nu şi mărimea sa în octeţi.

***NOTĂ:** Celor cinci tipuri de date definite în C, C++ adaugă încă două: **bool** şi **wchar_t**. Acestea sunt discutate în Partea a Doua a cărţii.*

Formatul exact al valorilor în virgulă mobilă va depinde de modul lor de introducere. Întregii vor corespunde în general mărimii normale a unui cuvânt pe calculatorul respectiv. Valorile de tip **char** sunt în general utilizate pentru a memora valori definite de setul de caractere ASCII. Valorile care ies din acest domeniu sunt tratate în mod diferit de diferite compilatoare.

Domeniul de cuprindere pentru **float** şi **double** va depinde de metoda folosită pentru a reprezenta numere în virgulă mobilă. Indiferent de metodă, domeniul este foarte cuprinzător. Standardul ANSI C specifică domeniul minim pentru valori în virgulă mobilă de la 1E-37 la 1E+37 (de la 10^{-37} la 10^{+37}). Numărul minim de cifre pentru precizie al fiecărui tip în virgulă mobilă este prezentat în **Tabelul 2-1**.

***NOTĂ:** Standardul propus pentru ANSI C++ nu specifică o mărime sau o sferă minimă pentru tipurile de bază. În schimb, el solicită pur şi simplu satisfacerea anumitor cerinţe. De exemplu, el spune că **int** „va avea domeniul de cuprindere normal pentru arhitectura sistemului". Totuşi, puteţi presupune că toate compilatoarele de C++ vor avea domeniile minime specificate de standardul ANSI C. Fiecare compilator de C++ specifică în fişierul antet **<climits>** mărimea şi domeniul de cuprindere al tipurilor de bază.*

Tip	Dimensiune aproximativă în biţi	Domeniu minimal de valori
char	8	de la -127 la 127
unsigned char	8	de la 0 la 255
signed char	8	de la -127 la 127
int	16	de la -32767 la 32767
unsigned int	16	de la 0 la 65535
signed int	16	Similar cu int
short int	16	Similar cu int
unsigned short int	16	de la 0 la 65535
signed short int	16	Similar cu short int
long int	32	de la -2.147.483.647 la 2.147.483.647
signed long int	32	Similar cu long int
unsigned long int	32	de la 0 la 4.294.967.295
float	32	Şase zecimale exacte
double	64	Zece zecimale exacte
long double	80	Zece zecimale exacte

Tabelul 2-1 *Toate tipurile de date definite prin standardul ANSI C*

Tipul **void** declară explicit că o funcţie nu returnează nici o valoare sau creează pointeri generici. Ambele cazuri sunt discutate în capitolele următoare.

Modificarea tipurilor de bază

Cu excepţia lui **void**, tipurile de bază pot fi precedate de diverşi specificatori de conversie. Un *specificator de conversie* se utilizează pentru a modifica tipul de bază pentru a se adapta mai precis la situaţii cât mai diverse. Iată lista specificatorilor de conversie:

signed
unsigned
long
short

Puteţi să aplicaţi specificatorii **signed, short** şi **unsigned** întregilor. Tipului de bază caracter i se pot aplica **signed** şi **unsigned**. De asemenea, puteţi aplica **long** pentru **double**.

Tabelul 2-1 prezintă toate combinaţiile de tipuri de date care corespund standardului ANSI C, împreună cu domeniile minimale şi mărimea aproximativă în biţi. (Aceste valori se aplică şi unui C++ tipic.)

Este permisă utilizarea lui **signed** pentru întregi, dar este redundantă, deoarece declararea implicită ca întreg presupune un număr cu semn. Utilizarea cea mai importantă pentru **signed** este pentru a modifica tipul **char** în implementările în care acesta este, implicit, fără semn.

Diferenţa dintre întregii cu şi fără semn constă în modul în care este interpretat bitul cu ordinul cel mai mare. Dacă specificaţi un întreg cu semn, compilatorul generează un cod care presupune că bitul de ordinul cel mai mare va fi utilizat ca *indicator pentru semn*. Dacă acesta este 0, numărul este pozitiv; dacă este 1, numărul este negativ.

În general, numerele negative sunt reprezentate utilizând *complementul lui 2*, care inversează toţi biţii din număr (cu excepţia bitului de semn), adună 1 la acest număr şi dă indicatorului pentru semn valoarea 1.

Întregii cu semn sunt importanţi în foarte mulţi algoritmi, dar ei ating doar jumătate din amplitudinea fraţilor lor fără semn. De exemplu, iată numărul 32.767:

0 1 1 1 1 1 1 1   1 1 1 1 1 1 1 1

Dacă bitul cu ordinul cel mai mare este 1, numărul va fi interpretat ca -1. Dar dacă îl declaraţi pe acelaşi ca un **unsigned int**, el devine 65.535 atunci când bitul respectiv are valoarea 1.

Nume de identificatori

În C/C++ numele variabilelor, ale funcţiilor, ale etichetelor şi ale altor diverse obiecte definite de către utilizator sunt numite *identificatori*. Aceşti identificatori pot să aibă unul sau mai multe caractere. Primul caracter trebuie să fie o literă sau o linie de subliniere, iar următoarele pot fi litere, cifre sau linia de sublinire. Iată câteva nume de identificatori corecte şi incorecte:

Corect	**Incorect**
numarator	1numarator
test23	salut!
bilant_mare	bilant...mare

Standardul ANSI C stipulează că identificatorii pot avea orice lungime. Totuşi, nu toate caracterele vor fi obligatoriu semnificative. Dacă identificatorul este implicat într-un proces de editare de legături externe, vor conta cel puţin şase caractere. Aceşti identificatori, denumiţi *nume externe*, includ numele funcţiilor şi ale variabilelor globale care aparţin mai multor fişiere. Dacă identificatorul nu este utilizat într-un proces de editare de legături din exterior, atunci vor fi semnificative cel puţin 31 de caractere. Acest tip de identificator este denumit *nume intern* şi include, de exemplu, nume de variabile locale. În C++ nu există limite ale lungimii unui identificator şi toate caracterele sunt semnificative. Această diferenţă poate fi importantă dacă doriţi să convertiţi un program din C în C++.

Într-un identificator literele mari sunt tratate distinct faţă de cele mici. Astfel, **numara**, **Numara** şi **NUMARA** sunt trei identificatori diferiţi.

Atât în C, cât şi în C++ un identificator nu poate fi identic cu un cuvânt-cheie şi nu trebuie să aibă acelaşi nume ca o funcţie din biblioteca C sau C++.

Variabile

După cum ştiţi probabil, *o variabilă* este numele unei locaţii din memorie utilizate pentru a păstra o valoare care poate fi modificată de program. Înainte de a le utiliza, variabilele trebuie declarate. Forma generală a unei declaraţii este:

tip listă_variabile;

Aici *tip* trebuie să fie un tip de dată valid, plus orice specificator de conversie, iar *listă_variabile* poate consta dintr-unul sau mai multe nume de identificatori, separate prin virgulă. Iată câteva declaraţii:

```
int i, j, l;
short int si;
unsigned int ui;
double bilant, profit, pierdere;
```



***REŢINEŢI:** În C/C++ numele variabilei nu are legătură cu tipul său.*

Unde se declară variabilele

Variabilele se declară, de obicei, în trei locuri: în interiorul funcţiilor, în cadrul definiţiei parametrilor funcţiei şi în afara oricărei funcţii. Avem, deci, de-a face cu variabile locale, parametri formali şi variabile globale.

Variabile locale

Variabilele care sunt declarate în interiorul unei funcţii sunt numite *variabile locale*. O parte din literatura de C/C++ numeşte aceste variabile *variabile*

automatice. Această carte foloseşte termenul mai uzual de „variabile locale". Variabilele locale nu sunt accesibile decât instrucţiunilor care sunt în interiorul blocului în care sunt declarate variabilele. Cu alte cuvinte, variabilele locale sunt cunoscute doar în interiorul propriului lor bloc de cod. Reţineţi că un bloc de cod începe cu o acoladă deschisă şi se termină cu o acoladă închisă.

Variabilele locale există doar atât cât se execută blocul de cod în care sunt declarate. Aceasta înseamnă că o variabilă locală este creată la începerea execuţiei blocului său şi este distrusă la încheiere.

Blocul de cod cel mai uzual în care sunt declarate variabilele locale este funcţia. De exemplu, să luăm următoarele două funcţii:

```
void func1(void)
{
    int x;
    x = 10;
}
void func2(void)
{
    int x;
    x = -199;
}
```

Variabila întreagă **x** este declarată de două ori, o dată în **func1()** şi o dată în **func2()**. **x** din **func1()** nu are nici o legătură şi nici un efect asupra variabilei **x** din **func2()**, deoarece fiecare **x** este cunoscut doar de codul din blocul în care este declarată variabila.

Limbajul C conţine cuvântul cheie **auto**, pe care îl puteţi utiliza pentru a declara variabile locale. Totuşi, deoarece toate variabilele care nu sunt globale se presupune că sunt implicit **auto**, cuvântul nu este utilizat, în principiu, niciodată. De aceea, exemplele din această carte nici nu îl vor conţine. (Se spune că **auto** a fost introdus în C pentru a asigura compatibilitatea cu predecesorul său, B. Tot aşa, **auto** este admis în C++ pentru a fi compatibil cu C.)

Din motive de convenienţă şi tradiţie, majoritatea utilizatorilor declară toate variabilele folosite de o funcţie imediat după acolada deschisă a funcţiei şi înainte de orice altă instrucţiune. Totuşi, puteţi să declaraţi variabile locale în orice bloc de cod. Blocul definit de o funcţie este pur şi simplu un caz particular. De exemplu:

```
void f(void)
{
    int t;
    scanf("%d", &t);

    if(t==1){
        char s[80];   /* aceasta este creata doar la
                         intrarea in acest bloc*/
        printf("introduceti numele:");
        gets(s);
        /*faceti ceva...*/
    }
}
```

Aici variabila locală **s** este creată la intrarea în blocul de cod **if** şi distrusă la ieşirea din el. Mai mult, **s** este cunoscută doar în interiorul blocului **if** şi nu este accesibilă din altă parte - nici chiar din alte zone ale funcţiei care o conţine.

Un avantaj de a declara o variabilă locală într-un bloc de condiţionare este că, pentru acea variabilă se va aloca memorie doar dacă va fi necesar, deoarece variabilele locale nu există până când nu se ajunge la blocul în care sunt declarate. Cu lipsa de memorie s-ar putea să vă confruntaţi când creaţi un cod pentru controlul anumitor procese (precum deschiderea uşii unui garaj care răspunde la un cod digital de siguranţă) care dispun de foarte puţin RAM.

Declararea de variabile în interiorul blocului care le utilizează ajută, de asemenea, la prevenirea efectelor secundare nedorite. Atâta vreme cât ele nu există în afara blocului în care au fost declarate, ele nu pot fi modificate accidental.

Există o diferenţă importantă între modul de declarare a variabilelor locale în C faţă de C++. În C, trebuie să declaraţi toate variabilele locale la începutul blocului în care le definiţi, înainte de orice instrucţiuni ale programului. De exemplu, următoarea funcţie este greşită dacă este compilată cu un compilator de C.

```
/* Aceasta functie este gresita daca este compilata cu un
   compilator de C, dar perfect acceptabila pentru un
   compilator de C++.
*/
void f(void)
{
    int i;
    i = 10;
    int j;   /*aceasta linie va determina o eroare*/
    j = 20;
}
```

Însă, în C++, această funcţie este perfect valabilă deoarece puteţi defini variabile locale în orice punct al programului. (Declararea de variabile în C++ este discutată în profunzime în Partea a Doua a acestei cărţi.)

Deoarece variabilele locale sunt create şi distruse la fiecare intrare, respectiv ieşire din blocul în care au fost declarate, conţinutul lor se pierde o dată cu

părăsirea blocului. Acest lucru este important să ni-l amintim când apelăm o funcţie. La apelarea ei, sunt create variabilele locale iar la încheierea ei acestea sunt distruse, ceea ce înseamnă că variabilele locale nu păstrează valorile între apelări. (Totuşi, puteţi determina compilatorul să păstreze aceste valori utilizând specificatorul de conversie **static**.)

Dacă nu se specifică altfel, variabilele locale sunt stocate în memoria stivă. Faptul că memoria stivă este o regiune dinamică şi în schimbare a memoriei explică de ce variabilele locale nu pot, în general, să păstreze valorile între apelările funcţiei.

Puteţi iniţializa o variabilă locală cu o valoare cunoscută. Această valoare va fi atribuită variabilei de fiecare dată când se va intra în blocul de cod în care este ea declarată. De exemplu, următorul program afişează numărul **10** de zece ori.

```
#include <stdio.h>
void f(void);
void main(void)
{
    int i;
    for(i=0; i<10; i++)  f();
}
void f(void)
{
    int j = 10;
    printf("%d ", j);
    j++;  /*aceasta linie nu are nici un efect */
}
```

Parametri formali

Dacă o funcţie urmează să folosească argumente, ea trebuie să declare variabilele pe care le acceptă ca valori ale argumentelor. Aceste variabile sunt denumite *parametri formali* ai funcţiei. Ele se comportă ca oricare altă variabilă locală din acea funcţie. Aşa cum se arată în următorul fragment de program, declararea lor are loc după numele funcţiei şi este închisă între paranteze:

```
/*Returneaza 1 daca c face parte din sirul s; daca nu, 0*/
este_in(char *s, char c)
{
    while(*s)
          if(*s==c) return 1;
          else s++;
    return 0;
}
```


Funcţia **este_in()** are doi parametri: **s** şi **c**. Această funcţie returnează 1 atunci când caracterul specificat în **c** este conţinut în şirul **s**; dacă nu, returnează 0.

Trebuie să specificaţi tipul parametrilor formali declarându-i aşa cum am arătat. Apoi puteţi să îi folosiţi în interiorul funcţiei ca variabile locale obişnuite. Reţineţi că, fiind variabile locale, ele sunt, de asemenea, dinamice şi sunt distruse la ieşirea din funcţie.

Ca şi pentru variabilele locale, puteţi să atribuiţi valori parametrilor formali ai funcţiei sau să îi folosiţi în expresii permise. Chiar dacă aceste variabile primesc valoarea argumentelor transmise funcţiei, puteţi să le folosiţi ca pe oricare altă variabilă locală.

Variabile globale

Spre deosebire de variabilele locale, *variabilele globale* sunt cunoscute în întreg programul şi pot fi utilizate de către orice zonă a codului. De asemenea, ele îşi vor păstra valoarea pe parcursul întregii execuţii a programului. Variabilele globale se creează prin declarare în afara oricărei funcţii. Orice expresie are acces la ele, indiferent de tipul blocului de cod în care se află expresia.

În următorul program variabila **contor** a fost declarată în afara oricărei funcţii. Chiar dacă declaraţia sa se află înaintea funcţiei **main()**, aţi fi putut să o plasaţi oriunde înainte de prima sa utilizare, dar nu într-o funcţie. Este bine, totuşi, să declaraţi variabilele globale la începutul programului.

```
#include <stdio.h>
int contor;  /*contor este global */
void func1(void);

void func2(void);
void main(void)

{
    contor = 100;

    func1();

}
void func1(void)
{
    int temp;
    temp = contor;

    func2();
```

```
    printf("contor este %d", contor);  /*va afisa 100 */
}
void func2(void)
{
    int contor;
    for(contor = 1; contor<10; contor++)

        putchar('.');
}
```

Priviți cu atenție acest program. Observați că, deși nici **main()** și nici **func1()** nu au declarat o variabilă locală cu numele **contor**, amândouă o utilizează. Totuși **func2()** a creat o variabilă locală cu numele **contor**. Când **func2()** se referă la **contor**, se referă doar la variabila locală, nu și la cea globală. Dacă o variabilă locală și una globală au același nume, toate referirile la numele variabilei din interiorul blocului în care a fost declarată variabila locală sunt pentru acea variabilă și nu au efect asupra variabilei globale. Acest lucru poate să ne convină dar, dacă uităm de el, programul poate să se comporte ciudat, chiar dacă el pare corect.

Stocarea variabilelor globale este făcută de compilator într-o anume regiune de memorie alocată acestui scop. Variabilele globale sunt utile atunci când mai multe funcții ale aceluiași program folosesc aceleași date. Totuși, ar trebui să evitați utilizarea variabilelor globale când nu este necesar. Ele ocupă spațiu în memorie tot timpul cât este executat programul, nu doar atunci când sunt necesare. În plus, utilizarea unei valori globale în locul uneia locale va face funcția mai puțin generală deoarece se bazează pe ceva care trebuie definit în afara ei. În sfârșit, utilizarea unui număr mare de variabile globale poate conduce la erori în program datorită efectelor secundare necunoscute și nedorite. O problemă majoră în realizarea de programe mari este schimbarea accidentală a valorii variabilei, în altă parte a programului, unde este, de asemenea, utilizată. Acest lucru se poate întâmpla foarte ușor în C/C++ dacă utilizați prea multe variabile globale în programe.

Modelatori de acces

Există doi modelatori care controlează modul de apelare sau modificare a variabilelor. Acești specificatori sunt **const** și **volatile.** Ei trebuie să preceadă specificatorii de tip și numele tipului de date la care se referă.

const

Variabilele de tip **const** nu pot fi modificate de programul dvs. (O variabilă de tip **const** poate să primească totuși o valoare inițială.) Compilatorul poate să plaseze variabilele de acest tip în Read Only Memory (ROM). De exemplu,


```
const int a=10;
```

creează o variabilă întreagă numită **a** cu valoarea inițială 10 pe care programul dvs. nu o poate modifica. Dar puteți să folosiți această variabilă în alte tipuri de expresii. O variabilă de tip **const** va primi valoarea ori dintr-o inițializare explicită, ori prin intermediul hard-ului.

Modelatorul **const** poate fi utilizat pentru a proteja obiectele indicate de argumentele unei funcții spre a nu fi modificate de acea funcție. Aceasta înseamnă că, atunci când un pointer este transmis unei funcții, acea funcție poate să modifice variabila indicată de pointer. Însă, dacă în declararea parametrilor pointerul este specificat ca fiind **const**, codul funcției nu va fi capabil să modifice obiectul indicat. De exemplu, funcția **linie_pt_spatiu()** din programul următor afișează o linie de unire pentru fiecare spațiu din argumentul său de tip șir, ceea ce înseamnă că șirul "acesta este un sir" va fi afișat astfel: "acesta-este-un-sir". Utilizarea modelatorului **const** în declararea parametrilor ne asigură că obiectul indicat de parametru nu poate fi modificat de codul din interiorul funcției.

```
#include <stdio.h>
void linie_pt_spatiu(const char *sir);
void main(void)
{
    linie_pt_spatiu("acesta este un test")
}
void linie_pt_spatiu(const char *str)
{
    while(*str)  {
          if(*str== ' ')printf("%c", '-');
          else printf("%c", *str);
          str++;
    }
}
```

Dacă ați fi scris **linie_pt_spatiu** astfel încât șirul să poată fi modificat, el nu ar fi compilat. De exemplu, dacă i-ați scris codul după cum urmează, veți primi o eroare în timpul compilării.

```
/*Acesta este un cod gresit*/
void linie_pt_spatiu(const char *str)
{
    while(*str)   {
          if(*str==' ') *str = '-';  /* nu pot face
                                        aceasta */
```

```
            printf("%c", *str);
            str++;
        }
    }
```

Multe funcții din biblioteca standard utilizează **const** în declarațiile lor de parametri. De exemplu, funcția **strlen()** are următorul prototip:

*size_t strlen(const char *sir);*

Specificând *sir* ca fiind o constantă, el nu va modifica șirul indicat. În general, atunci când o funcție din biblioteca standard nu trebuie să modifice un obiect indicat de un argument de apelare, parametrul este declarat ca fiind constant.

Puteți, de asemenea, să utilizați **const** pentru a verifica dacă programul dvs. nu modifică o variabilă. Rețineți că o variabilă de tip **const** poate fi modificată de ceva din afara programului. De exemplu, un element din hard poate să îi schimbe valoarea. Prin declararea unei variabile ca fiind de tip **const**, puteți dovedi că orice modificare a acesteia are loc din cauze externe.

volatile

Modelatorul **volatile** spune compilatorului că valoarea unei variabile poate să fie modificată pe căi nedeclarate explicit de program. De exemplu, adresa unei variabile globale poate fi transmisă rutinei ceasului sistemului de operare și utilizată pentru a păstra timpul real al sistemului, situație în care conținutul variabilei se modifică fără o instrucțiune de atribuire explicită. Acest lucru este important deoarece majoritatea compilatoarelor de C/C++ optimizează automat anumite expresii pe baza presupunerii că o variabilă rămâne neschimbată atât timp cât nu se află în partea stângă a unei instrucțiuni de atribuire, nefiind astfel nevoie să i se verifice conținutul la fiecare utilizare, iar alte compilatoare modifică ordinea evaluării unor expresii în timpul compilării. Modelatorul **volatile** blochează aceste intervenții.

Puteți utiliza **const** și **volatile** împreună. De exemplu, dacă 0x30 se presupune că este valoarea unui port care este modificată doar de condiții externe, următoarea declarație va împiedica orice posibilitate de apariție a efectelor secundare accidentale.

```
const volatile unsigned char *port=0x30;
```


Specificatori de clase de stocare

C admite patru specificatori de clase de stocare:

```
extern
static
register
auto
```

Acești specificatori spun compilatorului cum să stocheze variabilele care îi urmează. Specificatorul de stocare precede restul declarației de variabile. Forma sa generală este:

```
specificator_de_stocare tip nume_variab
```

extern

Deoarece C/C++ permit secțiunilor separate ale unui program mare să fie compilate independent și să li se editeze legăturile împreună, trebuie să existe un mod de a comunica tuturor fișierelor variabilele globale necesare programului. Deși C permite practic declararea unei variabile globale de mai multe ori, nu este bine să o faceți (poate crea probleme la editarea legăturilor). Și mai important este faptul că în C++ puteți declara o variabilă globală *doar o dată*. Cum veți informa atunci toate fișierele din program despre variabilele globale utilizate? Soluția este să declarați toate variabilele globale într-un singur fișier și să folosiți declarațiile **extern** în celelalte, ca în **Figura 2-1**.

În fișierul 2, lista de variabile globale a fost copiată din fișierul 1 iar declarațiilor le-a fost adăugat specificatorul **extern**. Acesta spune compilatorului că tipurile și

Fișier 1	Fișier 2
int x,y; char ch; main(void) { . . . } func1( ) { x=123; }	extern int x,y; extern char ch; func22(void) { x=y/10; } func23( ) { y=10; }

Figura 2-1 Folosirea variabilelor globale în module compilate separat

numele de variabile care îi urmează au fost declarate în altă parte. Cu alte cuvinte, **extern** oferă compilatorului informațiile despre numele și tipurile de variabile globale fără să creeze de fapt un nou loc de stocare a lor. Atunci când sunt editate legăturile celor două secțiuni, este rezolvat și accesul la variabilele externe.

Cuvântul cheie **extern** are forma generală:

```
extern lista_variab;
```

Mai există și o altă utilizare opțională pentru **extern** pe care o veți întâlni uneori. Când folosiți o variabilă globală în interiorul unei funcții, puteți să o declarați ca fiind de tip **extern**, așa cum este arătat aici:

```
int primul, ultimul;    /* definirea pentru primului si
                           ultimului ca variabile globale */
void main(void)
{
   extern int primul;    /* utilizarea optionala a
                            declaratiei extern */
.
.
.
}
```

Chiar dacă sunt permise declarări de variabile **extern**, așa cum ați văzut în exemplul anterior, ele nu sunt necesare. Atunci când compilatorul întâlnește o variabilă care nu a fost declarată în blocul respectiv, el verifică dacă acea variabilă este întâlnită în declarațiile de variabile din interiorul blocurilor care îl conțin pe acesta. Dacă nu o întâlnește, compilatorul verifică variabilele globale. Dacă întâlnește una, el presupune că trebuie să utilizeze variabila globală.

Variabile statice

Variabilele de tip **static** sunt variabile permanente în interiorul funcției sau fișierului în care se găsesc. Spre deosebire de variabilele globale, ele nu sunt cunoscute în afara funcției sau fișierului, dar ele își păstrează valoarea între două apelări. Această caracteristică este utilă când scrieți funcții generalizate și funcții de bibliotecă pe care le pot utiliza alte programe. **static** are efecte diferite asupra variabilelor locale și asupra celor globale.

Variabile locale statice

Atunci când aplicați specificatorul **static** unei variabile locale, compilatorul creează pentru ea un loc de stocare permanentă, similar celui rezervat unei variabile globale. Diferența esențială între variabilele locale statice și o variabilă globală este că cea locală statică rămâne cunoscută doar blocului în care a fost declarată. Mai simplu, o variabilă locală statică este o variabilă locală care își păstrează valoarea între apelările funcției.

Variabilele locale de tip **static** sunt foarte importante pentru crearea unor funcții de sine stătătoare, deoarece mai multe tipuri de rutine trebuie să păstreze valori între apelări. Dacă variabilele statice nu ar fi fost permise, ar fi fost utilizate variabile globale, riscându-se însă apariția efectelor secundare. Un exemplu de funcție care beneficiază de variabile locale statice este un generator de serii de numere care emite o valoare în funcție de cea precedentă. Ați putea folosi variabile globale pentru a reține această valoare, dar de fiecare dată când funcția este utilizată într-un program, ar trebui să declarați acea variabilă și să vă convingeți că nu va intra în conflict cu altă variabilă globală deja declarată. De asemenea, utilizarea unei variabile globale ar face această funcție greu de introdus într-o funcție de bibliotecă. Cea mai bună soluție este să declarați variabila care păstrează numărul generat ca **static**, ca în următorul fragment de program:

```
serii (void)
{

    static int serii_num;
    serii_num = serii_num+23;
    return serii_num;
}
```

În acest exemplu, variabila **serii_num** continuă să existe între apelările funcției și nu apare și dispare așa cum o fac variabilele locale obișnuite. Aceasta înseamnă că fiecare apelare a funcției **serii()** produce un nou număr al seriei bazat pe precedentul, fără să declare acea variabilă ca fiind globală.

Variabilei locale de tip **static** puteți să îi dați o valoare inițială. Această valoare îi este atribuită o singură dată - nu de fiecare dată când se intră în blocul de cod -, ca și în cazul variabilelor locale obișnuite. De exemplu, această versiune pentru **serii()** inițializează **serii_num** cu 100:

```
serii(void)
{
    static int serii_num = 100;
    serii_num = serii_num+23;
    return serii_num;
}
```

Așa cum arată funcția acum, seriile vor începe întotdeauna cu valoarea 123. Dacă lucrul este acceptabil pentru unele aplicații, majoritatea generatoarelor de

serii necesită specificarea de către utilizator a numărului inițial. O cale de a da variabilei **serii_num** o valoare specificată de utilizator este de a o declara variabilă globală și apoi de a-i da valoarea specificată. Dar exact aceasta am dorit să evităm atunci când am declarat-o statică. Aceasta ne conduce la cealaltă utilizare a specificatorului **static**.

Variabile globale statice

Aplicând specificatorul **static** unei variabile globale, vom spune compilatorului să creeze o variabilă globală care este cunoscută doar în fișierul în care a fost declarată. Aceasta înseamnă că, deși variabila este globală, rutinele din alte fișiere nu au acces le ea și nu îi pot modifica direct conținutul, ea fiind astfel protejată de efecte secundare. Pentru puținele situații pentru care o variabilă locală statică nu convine, puteți crea un mic fișier care conține doar funcția care are nevoie de variabila globală de tip **static**, pe care să-l compilați separat și să-l utilizați fără teamă de efecte secundare.

Pentru a ilustra o astfel de variabilă, exemplul de generator de numere din secțiunea precedentă a fost rescris, astfel încât o valoare primară să inițializeze seria printr-o apelare a unei funcții numite **incep_serie**. În continuare este prezentat întreg fișierul care conține **serii()**, **incep_serie()** și **serii_num:**

```
/* Toate acestea trebuie sa fie intr-un singur fisier. */
static int serii_num;
void incep_serie(int initial);
int serii(void);

serii(void)
{
    serii_num = serii_num+23;
    return serii_num;
}
/* initializeaza serii_num */
void incep_serie(int incep)
{
    serii_num = incep;
}
```

Introducând în **incep_serie()** o valoare întreagă se inițializează generatorul de serii. După aceasta se apelează **serii()** pentru a genera următorul element al seriei.

Să recapitulăm: numele variabilelor locale de tip **static** sunt cunoscute doar blocurilor de cod în care sunt declarate; numele variabilelor globale de tip **static** sunt cunoscute doar fișierului în care se găsesc. Dacă introduceți funcțiile **serii()** și **int_serie()** într-o bibliotecă, puteți utiliza funcțiile dar nu veți avea acces la variabila **serii_num** care este ascunsă restului codului din program. De fapt, puteți chiar să declarați și să utilizați o altă variabilă numită **serii_num** în programul dvs. (desigur, în alt fișier). În esență, specificatorul **static** permite existența unor variabile cunoscute doar funcțiilor care le necesită, fără să intre în conflict cu alte funcții.

Variabilele de tip **static** vă permit să ascundeți anumite secțiuni ale programului față de altele. Acesta poate fi un avantaj imens atunci când încercați să gestionați un program mare și complex.

Variabile de tip register

Specificatorul de stocare **register** se aplică prin tradiție doar variabilelor de tip **int** și **char**. Totuși standardul ANSI C îi dă definiția, astfel încât puteți să-l folosiți pentru oricare tip de variabilă.

Inițial, **register** cerea compilatorului să păstreze valoarea unei variabile într-un registru din CPU și nu în memorie, acolo unde sunt stocate variabilele în mod normal. Aceasta înseamnă că operațiile asupra unei variabile specificate cu **register**, păstrate în CPU, sunt executate mai repede decât asupra uneia obișnuite deoarece nu necesită acces la memorie pentru a determina sau a modifica valoarea ei.

Acum definiția specificatorului **register** a fost mult extinsă și el poate fi aplicat oricărui tip de variabilă. Standardul ANSI C declară simplu că „accesul la obiect este cât mai rapid posibil". (Standardul ANSI C++ stipulează că **register** este „o indicație dată compilatorului, că obiectul astfel declarat va fi utilizat din plin.") În practică, caracterele și întregii sunt în continuare stocați în registre ale CPU. Obiectele mari, cum sunt matricele, evident nu pot fi stocate într-un registru, dar ele pot totuși să beneficieze de un tratament preferențial din partea compilatorului. Variabilele **register** pot fi tratate diferit, astfel încât să corespundă modului de instalare a compilatorului de C/C++ și mediului său de operare. De fapt, tehnic este permis unui compilator să ignore **register** și să trateze normal variabilele specificate, dar aceasta se întâmplă rareori în practică.

Puteți să aplicați specificatorul **register** doar variabilelor locale și parametrilor formali ai unei funcții. Așadar, nu sunt permise variabile globale de tip **register**. Iată un exemplu care folosește variabile **register**. Această funcție calculează rezultatul lui M^e pentru valori întregi.

```
putere_int(register int m, register int e)
{
    register int temp;
    temp = 1;
    for(; e; e--) temp = temp * m;
    return temp;
}
```

În acest exemplu, **e**, **m** şi **temp** sunt declarate ca variabile de tip **register** deoarece sunt toate utilizate într-o buclă. Faptul că variabilele **register** sunt optimizate ca viteză le face ideale pentru controlul sau utilizarea în bucle. În general, variabilele tip **register** sunt utilizate acolo unde acţionează cel mai eficient, de cele mai multe ori în locurile în care se face foarte des referire la aceeaşi variabilă. Acest lucru este important de reţinut deoarece puteţi declara oricâte variabile ca fiind de tip **register**, dar nu toate vor beneficia de aceeaşi optimizare a vitezei de acces.

Numărul permis de variabile **register** care pot fi optimizate într-un bloc de cod este determinat atât de mediu cât şi de modul efectiv de instalare a limbajului C/C++. Nu trebuie să vă fie teamă de declararea prea multor variabile ca **register** deoarece compilatorul le transformă automat în tipul normal atunci când este atinsă o anumită limită. (Aceasta asigură portabilitatea codului pentru o gamă mare de procesoare.)

În mod normal pot fi păstrate în registrele CPU cel puţin două variabile **register** de tip **char** sau **int**. Deoarece mediile de instalare variază foarte mult, consultaţi manualul compilatorului pentru a afla dacă puteţi să aplicaţi şi alte tipuri de opţiuni de optimizare.

Deoarece o variabilă **register** poate fi stocată într-un registru CPU, aceste variabile nu au adrese, ceea ce înseamnă că nu veţi putea găsi adresa unei variabile **register** cu ajutorul operatorului & (discutat mai târziu în acest capitol).

Chiar dacă standardul ANSI C (şi cel propus pentru ANSI C++) a lărgit descrierea specificatorului **register** faţă de semnificaţia clasică, în practică el are un efect sesizabil doar asupra tipurilor întreg şi caracter. De aceea, nu veţi putea probabil să contaţi pe îmbunătăţirea substanţială a vitezei pentru alte tipuri de variabile.

Iniţializări de variabile

Puteţi să introduceţi o valoare într-o variabilă în momentul declarării, prin plasarea după numele variabilei a semnului egal şi a unei constante. Forma generală a iniţializării este:

```
tip nume_variab = constanta;
```

Iată câteva exemple:

```
char ch = 'a';
int prima = 0;
float bilant = 123.23;
```

Variabilele globale şi cele **statice** locale sunt iniţializate doar la începutul programului. Variabilele locale (în afara celor de tip **static**) sunt iniţializate de fiecare dată când este întâlnit blocul în care sunt declarate. Variabilele locale care nu sunt iniţializate au valori necunoscute înainte de prima atribuire pe care le-o efectuaţi. Variabilele globale şi locale de tip **static** neiniţializate sunt automat iniţializate cu 0.

Constante

Constantele se referă la valori fixe pe care programul nu poate să le modifice. Constantele pot fi de oricare din tipurile fundamentale de date. Modul în care este reprezentată fiecare constantă depinde de tipul său. Constantele de tip caracter sunt incluse între ghilimele simple. De exemplu, 'a ' şi '%' sunt ambele constante de tip caracter. C defineşte, de asemenea, caractere multioctet (mai ales în mediile care nu utilizează limba engleză.)

Constantele de tip întreg sunt specificate ca numere fără parte fracţionară. De exemplu, 10 şi -100 sunt constante de tip întreg. Constantele în virgulă mobilă cer punctul zecimal urmat de partea zecimală a numărului. De exemplu, 11.123 este o constantă în virgulă mobilă. C permite, de asemenea, utilizarea notaţiei ştiinţifice pentru numere în virgulă mobilă.

Există două tipuri de numere în virgulă mobilă: **float** şi **double**. Există, de asemenea, mai multe varietăţi ale tipurilor de bază pe care le puteţi obţine utilizând specificatorii de tip. Implicit, compilatorul C stabileşte pentru o constantă numerică cel mai scurt tip de date compatibil care o poate păstra. De aceea, 10 este implicit **int**, dar 60.000 este **unsigned**, iar 100.000 este **long**. Chiar dacă valoarea 10 poate intra în tipul **char**, compilatorul nu va transgresa limitele tipului. Singurele excepţii de la regula tipului celui mai scurt sunt constantele în virgulă mobilă, care sunt asimilate tipului **double**.

Compilatorul are setări implicite adecvate pentru majoritatea programelor pe care le veţi scrie. Totuşi, puteţi să specificaţi explicit tipul de constantă numerică pe care o doriţi utilizând un sufix. Pentru tipul în virgulă mobilă, dacă veţi scrie un F în urma sa, numărul va fi tratat ca fiind **float**. Dacă va fi urmat de L, numărul va deveni **long double**. Pentru tipul întreg, pentru **unsigned** veţi folosi sufixul U, iar pentru **long**, sufixul L. Iată câteva exemple:

Tip de dată	Exemple de constante
int	1, 123, 21000, -234
long int	35000L, -34L
short int	10, -1290
unsigned int	10000U, 987U, 40000
float	123.23F, 4.34e-3F
double	123.23, 12312333, -0.9876324
long double	1001.2L

Constante hexazecimale şi octale

Uneori este mai uşor să folosiţi un sistem de numeraţie în baza 8 sau 16 în loc de 10 (sistemul nostru zecimal standard). Sistemul de numeraţie bazat pe 8 este numit *octal* şi utilizează cifre de la 0 la 7. În octal, numărul 10 reprezintă 8 în zecimal. Sistemul de numeraţie în bază 16 este numit *hexazecimal* şi utilizează cifrele de la 0 la 9 plus literele de la A la F, care ţin locul elementelor 10, 11, 12, 13, 14 şi respectiv 15. De exemplu, numărul hexazecimal 10 este 16 în zecimal. Deoarece cele două sisteme de numeraţie sunt deseori întrebuinţate, C vă permite să specificaţi constantele întregi în hexazecimal sau octal în loc de zecimal. O constantă hexazecimală constă din 0x urmat de constanta în formă zecimală. O constantă octală începe cu 0. Iată câteva exemple:

```
int hex = 0x80;    /* 128 in zecimal */
int oct = 012;     /* 10 in zecimal */
```

Constante de tip şir

C admite şi un alt tip de constantă: şirul. Un *şir* este un set de caractere închise între ghilimele duble. De exemplu, "acesta este un test" este un şir. Aţi văzut exemple de şiruri în unele din instrucţiunile **printf()** din programele exemplu. Chiar dacă C vă permite să definiţi o constantă de tip şir, nu există efectiv un tip de date de tip şir.

Nu trebuie să confundaţi şirurile cu caracterele. O singură constantă de tip char este alcătuită dintr-un singur caracter şi încadrată între apostrofuri, ca de exemplu 'a'. Dar, "a" este un şir care conţine doar o singură literă.

Constante de tip backslash caracter

Încadrarea constantelor de tip caracter între ghilimele simple lucrează pentru majoritatea caracterelor afişabile. Însă altele, puţine la număr, cum ar fi caracterul de linie nouă, sunt imposibil de introdus de la tastatură. Din acest motiv, C include constante speciale *backslash caracter*.

C admite mai multe coduri backslash speciale (prezentate în **Tabelul 2-2**) astfel încât să puteţi introduce uşor caracterele respective ca pe nişte constante.

Pentru a asigura portabilitatea trebuie să folosiţi codurile backslash în locul codurilor lor ASCII.

De exemplu, următorul program emite o nouă linie şi un spaţiu de tabulare şi apoi afişează şirul **Acesta este un test.**

```
#include <stdio.h>
void main(void)
{
    printf("\n\tAcesta este un test");
}
```


Codul	**Semnificaţia**
\b	Backspace
\f	Avans hârtie
\n	Rând nou
\r	Retur de car
\t	Spaţiu de tabulare orizontal
\"	Ghilimele duble
\'	Ghilimele simple
\0	Nul
\\	Backslash (linie înclinată spre stânga)
\v	Spaţiu de tabulare vertical
\a	Alertă
\N	Constantă în octal (unde N este constanta în octal)
\xN	Constantă în hexazecimal (unde N este o constantă în hexazecimal)

Tabelul 2-2 *Coduri backslash*

Operatori

C conţine foarte mulţi operatori. De fapt, el acordă acestora o importanţă mai mare decât majoritatea limbajelor. C defineşte patru clase de operatori: *aritmetici*, *relaţionali*, *logici* şi *de acţiune pe biţi*. Pentru anumite sarcini C are operatori suplimentari.

Operatorul de atribuire

În C, puteţi să folosiţi operatorul de atribuire în cadrul oricărei expresii valide, lucru care nu este permis în majoritatea limbajelor de programare (inclusiv Pascal, BASIC şi FORTRAN), care tratează operatorul de atribuire ca pe un caz de instrucţiune specială. Forma generală a operatorului de atribuire este:

nume_variabilă = expresie;

unde o expresie poate fi o constantă simplă sau atât de complexă pe cât vă este necesar. Ca şi BASIC sau FORTRAN, C utilizează un singur semn egal pentru a indica atribuirea (spre deosebire de Pascal sau Modula2, care utilizează construcţia :=). *Ţinta*, sau membrul stâng al atribuirii, trebuie să fie o variabilă sau un pointer, nu o funcţie sau o constantă.

Frecvent, în literatura despre C şi în mesajele de eroare ale compilatorului veţi observa aceşti doi termeni: lvalue şi rvalue. Exprimat simplu, *lvalue* este orice obiect care poate să apară în partea din stânga a instrucţiunii de atribuire. În practică, în toate situaţiile, „lvalue" înseamnă „variabilă". Termenul *rvalue* se referă la expresiile din membrul drept al atribuirii şi, pur şi simplu, ea conţine valoarea unei expresii.

Conversii de tip la atribuire

Când variabilele de un anumit tip sunt amestecate cu variabile de alt tip, are loc o *conversie de tip*. Într-o instrucţiune de atribuire, regula de conversie este uşoară: valoarea din membrul drept (cel al expresiei) al declaraţiei de atribuire este convertită în tipul din membrul stâng (variabila ţintă), aşa cum se ilustrează în continuare:

```
int x;
char ch;
float f;

void func(void)
{
    ch = x;      /*linia 1*/
    x = f;       /*linia 2*/
    f = ch;      /*linia 3*/
    f = x;       /*linia 4*/
}
```

În linia 1, biţii de ordin mare din stânga ai variabilei întregi **x** sunt eliminaţi, rămânând pentru **ch** cei opt biţi de ordin mic. Dacă **x** este cuprins între 256 şi 0, **ch** şi **x** vor avea valori identice. Altfel, valoarea din **ch** va reflecta doar biţii de ordin mic ai lui **x**. În linia 2, **x** va primi partea nefracţionară a lui **f**. În linia 3, **f** va converti valoarea întreagă de 8 biţi din **ch** în aceeaşi valoare, dar în format de virgulă mobilă. Acelaşi lucru se întâmplă şi în linia 4, doar că **f** va converti o valoare întreagă în format de virgulă mobilă.

Când convertim din întregi în caractere şi din întregi lungi în întregi, se înlătură cantitatea corespunzătoare de biţi de ordin superior. În multe medii, aceasta înseamnă că se vor pierde 8 biţi atunci când vom trece de la un întreg la un caracter şi 16 când se va trece de la un întreg lung la un întreg.

Tipul destinaţiei	**Tipul expresiei**	**Posibile pierderi de informaţie**
signed char	char	Dacă valoarea >127, destinaţia este negativă
char	short int	Cei mai semnificativi opt biţi
char	int	Cei mai semnificativi opt biţi
char	long int	Cei mai semnificativi 24 de biţi
int	long int	Cei mai semnificativi 16 biţi
int	float	Partea zecimală şi posibil mai mult
float	double	Precizie, rezultat rotunjit
double	long double	Precizie, rezultat rotunjit

Tabelul 2-3 *Tipuri de conversie des folosite (presupunând cuvinte de 16 biţi)*

Tabelul 2-3 rezumă specificatorii de tip pentru atribuire. Reţineţi că prin conversiile **int** în **float**, sau **float** în **double** şi aşa mai departe, nu se câştigă precizie sau exactitate. Acest tip de specificatori schimbă doar forma de reprezentare a valorii. În plus, unele compilatoare (şi procesoare) C, întotdeauna când fac conversia unei variabile de tip **char** în **int** sau **float**, o tratează ca fiind pozitivă, indiferent de valoarea conţinută în ea. Alte compilatoare, atunci când convertesc variabile de tip **char,** tratează valorile mai mari decât 127 ca numere negative. În general, puteţi să folosiţi variabile de tip **char** pentru caractere şi să utilizaţi **int**, **short int** sau **signed char** atunci când doriţi să evitaţi problemele de portabilitate.

Pentru a utiliza **Tabelul 2-3** pentru conversiile neprezentate, convertiţi pe rând, de la un tip la altu!, până la sfârşit. De exemplu, pentru a face conversia din **double** în **int**, mai întâi convertiţi **double** în **float** şi apoi din **float** în **int**.

Atribuiri multiple

C vă permite să atribuiţi aceeaşi valoare mai multor variabile prin utilizarea atribuirii multiple într-o singură instrucţiune de atribuire. De exemplu, acest fragment de program atribuie valoarea 0 lui **x**, **y** şi **z**:

```
x = y = z = 0;
```

În programele profesionale, această metodă este des utilizată pentru a atribui variabilelor valori uzuale.

Operatori aritmetici

Tabelul 2-4 prezintă operatorii aritmetici din C. În C, operatorii +, -, * şi / lucrează la fel ca în majoritatea altor limbaje. Puteţi să îi aplicaţi oricăror tipuri de date permise în C. Când aplicaţi / unui întreg sau unui caracter, orice rest va fi eliminat. De exemplu, 5/2 va fi egal cu 2 pentru o împărţire întreagă.

Operatorul modulo, %, lucrează în C ca şi în alte limbaje, reţinând restul unei împărţiri de întregi. Nu se poate folosi pentru valori în virgulă mobilă. Următorul fragment de cod prezintă utilizarea lui %:

```
int x, y;

x = 5;
y = 2;

printf("%d", x/y);   /* va afisa 2 */
printf("%d", x%y);   /* va afisa 1, restul impartirii de
                        intregi */

x = 1;
y = 2;

printf("%d %d", x/y, x%y);    /*  va afisa 0 1 */
```

Ultima linie afişează un 0 şi un 1 deoarece 1/2 într-o împărţire de întregi este 0, cu restul 1.

Minusul unar înmulţeşte elementul cu -1, deoarece orice număr precedat de un semn minus îşi schimbă semnul.

Operator	Acţiune
-	Scădere, de asemenea şi minus unar
+	Adunare
*	Înmulţire
/	Împărţire
%	Modul
--	Decrementare
++	Incrementare

Tabelul 2-4. *Operatori aritmetici*

Increment şi decrement

C include doi operatori utili care nu se găsesc în general în alte limbaje. Ei sunt operatorii de incrementare şi de decrementare, ++ şi --. Operatorul ++ adună 1 la variabila operată iar -- scade 1. Cu alte cuvinte:

```
x = x+1;
```

este acelaşi lucru cu:

```
++x;
```

şi

```
x = x-1;
```

este acelaşi lucru cu:

```
x--;
```

Atât operatorii de incrementare cât şi cei de decrementare pot să fie plasaţi înainte sau după variabila operată. De exemplu,

```
x = x+1;
```

poate fi scris

```
++x;
```

sau

```
x++;
```

Există totuşi o diferenţă între forma cu prefix şi cea cu sufix când utilizaţi aceşti operatori într-o expresie. Atunci când operatorul de incrementare sau de decrementare precede variabila, C efectuează operaţiile respective înainte de a obţine valoarea variabilei pentru a fi utilizată în expresie. Dacă operatorul urmează variabilei, C obţine valoarea variabilei înainte de incrementare sau de decrementare. De exemplu,

```
x = 10;
y = ++x;
```

atribuie valoarea 11 lui **y**. Dacă veţi scrie codul astfel:

```
x = 10;
y = x++;
```

lui **y** i se va atribui 10. În ambele moduri, lui **x** i se atribuie valoarea 11; diferenţa constă în momentul efectuării operaţiei.

Majoritatea compilatoarelor C/C++ produc rapid un cod obiect eficient pentru operaţiile de incrementare şi de decrementare - cod care este mai bun decât cel obţinut prin utilizarea instrucţiunii de atribuire echivalente. Din acest motiv, ar trebui să utilizaţi aceşti operatori de câte ori puteţi.

Iată ordinea de precedenţă a operatorilor aritmetici:

De ordinul cel mai înalt	++ --
	- (minus unar)
	* / %
De ordinul cel mai coborît	+ -

Operatorii cu aceeaşi precedenţă sunt evaluaţi de compilator de la stânga la dreapta. Desigur, puteţi să utilizaţi paranteze pentru a modifica ordinea execuţiei. C tratează parantezele la fel cum le tratează şi toate celelalte limbaje. Ele forţează ca o operaţie sau un set de operaţii să aibă un nivel de precedenţă mai înalt.

Operatori relaţionali şi logici

În termenul „operator relaţional", *relaţional* se referă la relaţiile pe care aceste valori pot să le aibă cu altele. În termenul „operator logic", *logic* se referă la modul în care sunt corelate aceste relaţii. Deoarece operatorii relaţionali şi cei logici lucrează de obicei împreună, ei sunt discutaţi şi aici tot împreună.

Operatorii relaţionali şi cei logici se bazează pe ideea de adevărat şi fals. În C, adevărat este orice valoare care diferă de 0. Fals este 0. Expresiile care utilizează operatori relaţionali şi logici returnează 0 pentru fals şi 1 pentru adevărat.

Tabelul 2-5 prezintă aceşti operatori. Tabela de adevăr a operatorilor logici utilizează 0 şi 1.

p	q	p&&q	p||q	!p
0	0	0	0	1
0	1	0	1	1
1	1	1	1	0
1	0	0	1	0

Atât operatorii relaţionali cât şi cei logici au o precedenţă mai scăzută decât cei aritmetici. Aceasta înseamnă că o expresie ca 10>1+12 este evaluată ca şi cum ar

Operatori relaţionali

Operator	Acţiune
>	Mai mare decât
>=	Mai mare sau egal
<	Mai mic decât
<=	Mai mic sau egal
==	Egal
!=	Diferit

Operatori logici

Operator	Acţiune
&&	AND (ŞI)
||	OR (SAU)
!	NOT (NEGAT)

Tabelul 2-5. *Operatori relaţionali şi logici*

fi fost scrisă 10>(1+12). Desigur, rezultatul este fals.

După cum vedeţi, puteţi să combinaţi mai multe operaţii într-o expresie:

```
10>&&!(10<9)||3<=4
```

În acest caz, rezultatul este adevărat.

Chiar dacă nici C şi nici C++ nu conţin operatorul logic SAU exclusiv (XOR), puteţi să creaţi foarte uşor o funcţie care să execute această sarcină, utilizând ceilalţi operatori logici cunoscuţi. Rezultatul unei operaţii XOR este adevărat dacă şi numai dacă un element (dar nu ambii) este adevărat. Următorul program conţine funcţia **xor()**, care returnează rezultatul unei operaţii cu SAU exclusiv operând asupra celor două argumente proprii.

```
#include <stdio.h>

int xor(int a, int b);

void main(void)
{
    printf("%d", xor(1, 0));
    printf("%d", xor(1, 1));
    printf("%d", xor(0, 1));
    printf("%d", xor(0, 0));
}
```

```
/* efectuarea unei operatii XOR utilizand cele doua
   argumente. */
xor(int a, int b)
{
    return (a || b) && !(a && b);
}
```

Următorul tabel arată prioritățile relative pentru operatorii relaționali și cei logici.

De ordinul cel mai înalt	!
	> >= < <=
	== !=
	&&
De ordinul cel mai coborît	||

Ca și în expresiile aritmetice, puteți utiliza parantezele pentru a modifica ordinea firească de evaluare a unei expresii relaționale și/sau logice. De exemplu,

```
!0&&0||0
```

este fals. Dar dacă adăugați paranteze acestei expresii, așa cum vedeți mai jos, rezultatul este adevărat:

```
!(0&&0)||0
```

Rețineți că toate expresiile relaționale și logice au ca rezultat 0 sau 1. De aceea, următorul fragment de program este corect și va afișa cifra 1.

```
int x;

x = 100;
printf("%d", x>10);
```

Operatori de acțiune pe biți

Spre deosebire de multe alte limbaje, C admite un complement deplin cu operatori cu acțiune pe biți. Deoarece C a fost proiectat să ia locul limbajului de asamblare pentru majoritatea sarcinilor, el trebuie să fie capabil să asigure multe operații care pot fi executate în asamblor, inclusiv operații asupra biților. *Operațiile asupra biților* se referă la testare, inițializare sau deplasare a biților existenți într-un octet sau într-un cuvânt care corespunde tipurilor de date **char** și **int** și variantelor acestora din standardul de C. Nu puteți să le utilizați asupra tipurilor **float**, **double**, **long**

double, **void** sau asupra altor tipuri mai complexe. **Tabelul 2-6** prezintă operatorii care se aplică biților individuali ai variabilelor.

Operatorii pentru biți AND, OR și NOT (complement) au aceleași tabele de adevăr ca și echivalentele lor logice, doar că ei lucrează bit cu bit. SAU exclusiv are următoarea tabelă de adevăr:

p	q	p^q
0	0	0
1	0	1
1	1	0
0	1	1

Așa cum arată tabelul, rezultatul operației XOR este adevărat dacă numai unul dintre termeni este adevărat; altfel, el este fals.

Operațiile cu biți își găsesc adesea aplicații în cadrul driverelor de dispozitiv - așa cum sunt programele pentru modem, rutinele fișierelor de pe disc și rutinele de tipărire - deoarece pot fi utilizate pentru a descoperi anumiți biți, așa cum este cel de paritate. (Bitul de paritate confirmă că ceilalți biți din acel octet sunt nemodificați. El este de obicei bitul cu cel mai mare ordin al fiecărui octet.)

Gândiți operatorul pentru biți AND ca un mod de a curăța un bit. Aceasta înseamnă că orice bit care este 0 într-unul din termeni determină bitul corespunzător al rezultatului să fie 0. De exemplu, următoarea funcție citește un caracter din portul modemului utilizând funcția **citeste_modem()** și reinițializează bitul de paritate la 0.

```
char preia_char_din_modem(void)
{
    char ch;

    ch = citeste_modem(); /* preia un caracter din portul
                              modemului */
```

Operator	Acțiune
&	AND
|	OR
^	OR exclusiv (XOR)
~	Complement față de 1 (NOT)
>>	Deplasare la dreapta
<<	Deplasare la stânga

Tabelul 2-6. Operatori pentru biți

```
    return(ch & 127);
}
```

Paritatea este deseori indicată de cel de-al optulea bit, care este inițializat cu 0, prin operația AND cu un octet care are primii 7 biți inițializați cu 1 iar bitul 8 cu 0. Expresia **ch & 127** înseamnă că biții din **ch** sunt combinați prin operatorul AND cu cei care formează numărul 127. Rezultatul efectiv este acela că al optulea bit din **ch** capătă valoarea 0. În următorul exemplu, se presupune că **ch** a primit caracterul **A** și că are și bitul de paritate stabilit la 1.

```
Bit de paritate
↓
  11000001      ch conține un „A" cu bit de paritate
  01111111      127 în binar
& ________      AND asupra biților
  01000001      „A" fără bit de paritate
```

Operatorul pentru biți OR, ca opus al lui AND, poate fi utilizat pentru a atribui unui bit valoarea 1. Orice bit inițializat cu 1 în orice termen determină ca bitul corespunzător al rezultatului să fie 1. Următorul exemplu reprezintă 128 | 3.

```
  10000000      128 în binar
  00000011      3 în binar
| ________      OR asupra biților
  10000011      rezultat
```

Un OR exclusiv, prescurtat uzual cu XOR, va inițializa un bit dacă și numai dacă biții pe care îi compară sunt diferiți. De exemplu, 127^120 este:

```
  01111111      127 în binar
  01111000      120 în binar
^ ________      XOR în logica biților
  00000111      rezultat
```

REȚINEȚI: *operatorii relaționali și cei logici determină întotdeauna un rezultat care este 0 sau 1, în timp ce operatorii similari pentru biți determină o valoare arbitrară în concordanță cu operația respectivă. Cu alte cuvinte, operațiile pentru biți pot să ducă la alte valori decât 0 și 1, în timp ce operatorii logici evaluează întotdeauna la 0 sau 1.*

Operatorii de deplasare pentru biți, >> și <<, deplasează toți biții dintr-o

variabilă la dreapta sau la stânga după cum se specifică. Forma generală a instrucțiunii de deplasare la dreapta este:

variabilă>>număr de poziții ale biților

Forma generală a instrucțiunii pentru deplasarea la stânga este:

variabilă<<număr de poziții ale biților

Deoarece biții sunt mutați către un capăt, la celălalt capăt se adaugă zerouri. (În cazul unui întreg negativ cu semn o deplasare la dreapta va determina introducerea unui 1, astfel încât bitul de semn se va păstra.) Țineți minte că o deplasare nu este o rotație. Biții deplasați dincolo de capăt nu se întorc la capătul celălalt, ci sunt sunt pierduți.

Operațiile de deplasare a biților pot fi foarte utile atunci când decodificați intrarea de la un dispozitiv extern, precum un convertor D/A și citiți informațiile de stare. Operatorii de deplasare pentru biți pot, de asemenea, să înmulțească și să împartă întregi. O deplasare la dreapta împarte efectiv un număr cu 2 iar o deplasare la stânga îl înmulțește cu 2, așa cum se vede în **Tabelul 2-7**. Următorul program ilustrează operatorii de deplasare.

```
/* Un exemplu de deplasare pe biti */
#include <stdio.h>

void main(void)
{
    unsigned int i;
    int j;

    i = 1;

    /* deplasare la stanga */
    for(j=o; j<4; j++) {
        i = i << 1;    /* deplasare la stanga a lui i cu 1,
                          care este acelasi lucru cu
                          inmultirea cu 2 */
        printf("deplasare la stanga %d: %d\n", j, i);
    }

    /*deplasare la dreapta*/
    for(j=o; j<4; j++) {
        i = i >> 1;    /* deplasare la dreapta a lui i cu 1,
                          care este acelasi lucru cu
```

```
                    impartirea cu 2 */
          printf("deplasare la dreapta %d: %d\n", j, i);
    }
}
```

Operatorul de complement al lui 1, ~, inversează valoarea fiecărui bit dintr-un octet. Cu alte cuvinte, 1 este transformat în 0 iar 0 în 1.

Operatorii pentru biţi sunt deseori utilizaţi în rutine criptate. Dacă doriţi ca un fişier de pe disc să fie ilizibil, efectuaţi asupra lui câteva operaţii cu biţi. Una dintre cele mai simple metode este cea de complementare a fiecărui octet utilizând complementul lui 1, care inversează fiecare bit al octetului, aşa cum se prezintă mai jos:

Octetul original	00101100	Acelaşi
După prima complementare	11010011	
După a doua complementare	00101100	

Notaţi că o secvenţă de două complementări succesive produce numărul original. Astfel, primul complement reprezintă versiunea codificată a acelui octet. Al doilea complement decodează octetul la valoarea sa originală.

Pentru a codifica un caracter puteţi folosi funcţia **encode()**:

unsigned char x;	x după executarea fiecărei instrucţiuni	valoarea lui x
x=7;	0 0 0 0 0 1 1 1	7
x=x<<1;	0 0 0 0 1 1 1 0	14
x=x<<3;	0 1 1 1 0 0 0 0	112
x=x<<2;	1 1 0 0 0 0 0 0	192
x=x>>1	0 1 1 0 0 0 0 0	96
x=x>>2	0 0 0 1 1 0 0 0	24

Fiecare deplasare la stânga determină o înmulţire cu 2.
Remarcaţi că s-a pierdut informaţie după x<<2, fiindcă un bit a depăşit marginea din stânga.

Fiecare deplasare la dreapta determină o împărţire cu 2. Remarcaţi că împărţirile ulterioare nu readuc biţii pierduţi.

Tabelul 2-7. *Înmulţirea şi împărţirea folosind operatorii de deplasare*


```
/* O functie de codificare. */
char encode(char ch)
{
    return(~ch);   /*il complementeaza*/
}
```

Operatorul ?

C conţine un operator foarte puternic şi util care înlocuieşte anumite instrucţiuni de forma dacă-atunci-altfel. Operatorul ternar **?** are forma generală :

Exp1 ? Exp2 : Exp3;

unde *Exp1, Exp2 şi Exp3* sunt expresii. Reţineţi utilizarea şi amplasarea celor două puncte.

Operatorul **?** lucrează astfel: se evaluează *Exp1*. Dacă este adevărată se evaluează *Exp2* şi valoarea ei se atribuie expresiei. Dacă *Exp1* este falsă, se evaluează *Exp3* şi valoarea sa devine valoarea expresiei. De exemplu, în:

```
x = 10;
y = x>9 ? 100 : 200;
```

lui **y** i se atribuie valoarea 100. Dacă **x** ar fi fost mai mic decât 9, **y** ar fi primit valoarea 200. Acelaşi cod scris utilizând instrucţiunile **if-else** este:

```
x = 10;

if(x>9) y = 100;
else y = 200;
```

Operatorul ? va fi discutat mai amănunţit în **Capitolul 3** în legătură cu alte instrucţiuni de condiţionare în C.

Operatorii & şi *

Un *pointer* este adresa din memorie a unei variabile. O *variabilă de tip pointer* este o variabilă declarată explicit pentru a reţine un pointer către un obiect de un tip specificat. Cunoaşterea adresei unei variabile poate fi de mare ajutor în anumite tipuri de rutine. Pointerii au în C trei funcţii principale. Ei pot să asigure o cale rapidă de acces la elementele unei matrice; permit funcţiilor în C să modifice parametrii de apelare; în sfârşit, asigură funcţionarea listelor înlănţuite şi altor structuri de date dinamice. **Capitolul 5** este dedicat în exclusivitate pointerilor, în

capitolul de față fiind prezentați pe scurt cei doi operatori care sunt utilizați pentru a lucra cu pointeri.

Primul operator pentru pointeri este &, un operator unar care returnează adresa din memorie a unui element. (Rețineți că un operator unar cere un singur element asupra căruia operează.) De exemplu,

```
m = &numara;
```

plasează în **m** adresa din memorie a variabilei **numara**. Această adresă este locația variabilei în memoria internă a calculatorului. Ea nu are nici o legătură cu valoarea din **numara**. Puteți să considerați & ca reprezentând „adresa lui...". De aceea, instrucțiunea de atribuire precedentă înseamnă „**m** primește adresa lui **numara**".

Pentru o mai bună înțelegere a acestei atribuiri, să presupunem că variabila **numara** se găsește în locația de memorie 2000. Să mai presupunem că **numara** are valoarea 100. Atunci, după atribuirea precedentă, **m** va conține valoarea 2000.

Al doilea operator pentru pointeri este *, complementul lui &. Operatorul * este unar și returnează valoarea din variabila localizată la adresa specificată. De exemplu, dacă **m** conține adresa din memorie a variabilei **numara**

```
q = *m;
```

plasează valoarea din **numara** în **q**. Acum **q** are valoarea 100 deoarece 100 este stocat în locația 2000, adresa din memorie care a fost stocată în **m**. Considerați * ca reprezentând „din adresa". În acest caz, ați putea să citiți instrucțiunea astfel: „**q** primește valoarea din adresa **m**".

Din nefericire, simbolurile pentru înmulțire și pentru „din adresa" sunt aceleași, iar simbolul pentru AND asupra biților este același cu cel pentru „adresa lui...". Acești operatori nu au nici o legătură unii cu ceilalți. Atât & cât și * sunt executați cu prioritate față de toți ceilalți operatori aritmetici cu excepția lui minus unar, care are aceeași precedență.

Variabilele care vor păstra pointeri trebuie declarate ca atare. Variabilele care vor păstra adrese de memorie, sau pointeri, cum sunt numiți în C, trebuie să fie declarate plasând * în fața numelui de variabilă. Aceasta îi spune compilatorului că ea va păstra un pointer către acel tip de variabilă. De exemplu, pentru a declara **ch** ca un pointer către un caracter, veți scrie:

```
char *ch;
```

Aici, **ch** nu este un caracter, ci un pointer către un caracter - este o mare diferență. Tipul de date către care indică un pointer, în acest caz **char**, este numit *tipul de bază* al pointerului. Dar variabila pointer însăși este o variabilă care păstrează adresa unui obiect de acel tip de bază. Astfel, un pointer de tip caracter

(sau oricare altul) este de mărime suficientă pentru a păstra o adresă, așa cum este ea definită de arhitectura calculatorului pe care rulează. Însă rețineți că un pointer trebuie să indice doar către date care sunt de tipul de bază al pointerului.

În aceeași instrucțiune de declarare puteți amesteca variabile de tip pointer cu variabile obișnuite. De exemplu:

```
int x, *y, numara;
```

declară **x** și **numara**, variabile de tip întreg, iar **y** ca pointer către una de tip întreg.

Următorul program utilizează operatorii * și & pentru a introduce valoarea 10 în variabila numită **tinta**. După cum vă așteptați, acest program afișează pe ecran valoarea 10.

```
#include <stdio.h>

void main(void)
{
    int tinta, sursa;
    int *m;

    sursa = 10;
    m = &sursa;
    tinta = *m;

    printf("%d", tinta);
}
```

Operatorul din timpul compilării, *sizeof*

sizeof este un operator unar utilizat în timpul compilării care returnează lungimea în octeți a variabilei sau a specificatorului de tip dintre paranteze care îi urmează. De exemplu, presupunând că întregii au 2 octeți iar tipul **float** are 8,

```
float f;
printf("%d ", sizeof f);
printf("%d ", sizeof(int));
```

va afișa **8 2**.

Rețineți că pentru a calcula mărimea (size) unui tip, trebuie să includeți numele tipului între paranteze. Acest lucru nu este necesar pentru numele de variabile, deși nu este greșit.

C definește (utilizând **typedef**) un tip special numit **size_t**, care corespunde

aproximativ unui întreg fără semn. Practic, valoarea returnată de **sizeof** este de tipul **size_t**. În practică însă, puteţi să îl consideraţi (şi să îl utilizaţi) ca şi cum ar fi o valoare întreagă fără semn.

În primul rând **sizeof** ne ajută să creăm un cod portabil care depinde de mărimea tipurilor de date construite în C. De exemplu, imaginaţi-vă un program de baze de date care necesită să stocheze şase valori întregi în fiecare înregistrare. Dacă doriţi să utilizaţi acest program de baze de date pe o diversitate de calculatoare, nu puteţi stabili dvs. mărimea unui întreg, ci trebuie să determinaţi mărimea sa efectivă utilizând **sizeof**. Aşa stând lucrurile, aţi putea folosi următoarea rutină pentru a scrie o înregistrare într-un fişier pe disc.

```
/ *Scrieti 6 intregi intr-un fisier pe disc. */
void preia_inreg(int inreg[6], FILE *fp)
{
    int lung;

    lung = fwrite(inreg, sizeof inreg, 1, fp);
    if(lung != 1) printf("eroare de scriere");
}
```

Aşa cum este scris, **preia_inreg** este compilat şi rulat corect pe orice calculator, indiferent de numărul de octeţi conţinuţi într-un întreg.

Pentru a încheia, **sizeof** este evaluată în momentul compilării, iar valoarea pe care o determină este tratată ca o constantă în cadrul programului dvs.

Operatorul virgulă

Operatorul virgulă asigură înşiruirea mai multor expresii. Partea din stânga operatorului virgulă este întotdeauna evaluată ca **void**. Aceasta înseamnă că expresia din dreapta stabileşte valoarea întregii expresii separate prin virgulă. De exemplu,

```
x = (y=3, y+1);
```

în primul rând atribuie lui **y** valoarea **3** şi apoi atribuie lui **x** valoarea 4. Parantezele sunt necesare deoarece operatorul virgulă nu are prioritate faţă de operatorul de atribuire.

În principal, virgula stabileşte o succesiune de operaţii. Atunci când o utilizaţi în partea dreaptă a instrucţiunii de atribuire, valoarea atribuită este cea a ultimei expresii din lista separată prin virgule.

Operatorul virgulă are ceva din înţelesul cuvântului „şi" din limba vorbită, aşa cum este utilizat în propoziţia „fă asta, şi asta, şi asta".

Operatorii punct (.) şi săgeată (->)

Operatorii . (punct) şi -> (săgeată) se referă la elemente individuale ale structurilor şi uniunilor. *Structurile* şi *uniunile* sunt tipuri de date agregate la care se poate avea acces sub un singur nume (a se vedea **Capitolul 7**).

Operatorul punct este utilizat atunci când lucrăm cu structuri sau uniuni efective. Operatorul săgeată este folosit împreună cu un pointer la o structură sau la o uniune. De exemplu, având următorul fragment de program:

```
struct angajati
{
    char nume[80];
    int virsta;
    float salariu;
} angaj;
struct angajati *p = &angaj;    /* adresa lui angaj in p */
```

aţi putea scrie următorul cod pentru a atribui valoarea 123,23 elementului **salariu** din variabila de tip structură **angaj**.

```
angaj.salariu = 123.23;
```

Aceeaşi atribuire utilizând un pointer la **angaj** ar fi:

```
p->salariu = 123.23;
```

Operatorii ( ) şi [ ]

Parantezele rotunde sunt operatori care măresc prioritatea operaţiilor din interiorul lor. În parantezele pătrate se înscriu indicii matricelor (acestea vor fi discutate detaliat în **Capitolul 4**). Având o matrice, expresia din parantezele pătrate este un indice al acelei matrice. De exemplu,

```
#include <stdio.h>
char s[80];

void main(void)
{
    s[3] = 'X';
    printf("%c" , s[3]);
}
```

atribuie mai întâi valoarea **'X'** celui de-al patrulea element (rețineți că toate matricele încep în C de la 0) al matricei **s** și apoi afișează acel element.

Rezumatul priorităților

Tabelul 2-8 prezintă prioritatea operațiilor în C. Rețineți că toți operatorii, cu excepția operatorilor unari și a operatorului ?, operează de la stânga la dreapta. Operatorii unari (*, &, -) și ? operează de la dreapta la stânga.

> ***NOTĂ:** C++ definește câțiva operatori suplimentari, care vor fi discutați pe larg în Partea a doua.*

Expresii

Operatorii, constantele și variabilele sunt componentele expresiilor. O *expresie* în C este orice combinație validă formată din aceste elemente. Deoarece majoritatea expresiilor tind să urmeze regulile generale din algebră, echivalența este deseori generalizată. Totuși, câteva aspecte ale expresiilor sunt specifice pentru C (și C++).

Ordinea evaluării

Nici standardul ANSI C și nici cel propus pentru ANSI C++ nu specifică ordinea în care sunt evaluate subexpresiile dintr-o expresie. Ele lasă la latitudinea

<table>
<tr><td>Precedență maximă</td><td>( ) [ ] -> .</td></tr>
<tr><td></td><td>! ~ ++ -- - (de tip) * & sizeof</td></tr>
<tr><td></td><td>* / %</td></tr>
<tr><td></td><td>+ -</td></tr>
<tr><td></td><td><< >></td></tr>
<tr><td></td><td>< <= > >=</td></tr>
<tr><td></td><td>== !=</td></tr>
<tr><td></td><td>&</td></tr>
<tr><td></td><td>^</td></tr>
<tr><td></td><td>|</td></tr>
<tr><td></td><td>&&</td></tr>
<tr><td></td><td>||</td></tr>
<tr><td></td><td>?:</td></tr>
<tr><td></td><td>= += -= *= /= etc.</td></tr>
<tr><td>Precedență minimă</td><td>,</td></tr>
</table>

Tabelul 2-8. *Prioritatea operatorilor în C*

compilatorului rearanjarea unei expresii pentru a elabora codul optim. Aceasta înseamnă însă că un cod nu va trebui să se bazeze pe ordinea în care vor fi evaluate subexpresiile. De exemplu, expresia:

```
x = f1() + f2();
```

nu ne asigură că **f1()** va fi apelată înainte de **f2()**.

Conversia automată în expresii

Atunci când într-o expresie sunt amestecate constante și variabile de diferite tipuri, ele sunt convertite în același tip. Compilatorul face conversia tuturor elementelor asupra cărora se operează în tipul celui mai mare, acțiune numită *promovarea tipului*. Mai întâi, toate valorile de tip **char** și **short int** sunt automat evaluate ca **int**. (Acest proces este numit *promovarea la întreg*.) O dată încheiat acest proces, toate celelalte conversii sunt efectuate operație cu operație, așa cum este descris în următorul algoritm de conversie a tipului:

DACĂ un element este **long double**
ATUNCI următorul este convertit în **long double**
ALTFEL DACĂ un element este **double**
ATUNCI următorul este convertit în **double**
ALTFEL DACĂ un element este **float**
ATUNCI următorul este convertit în **float**
ALTFEL DACĂ un element este **unsigned long**
ATUNCI următorul este convertit în **unsigned long**
ALTFEL DACĂ un element este **long**
ATUNCI următorul este convertit în **long**
ALTFEL DACĂ un element este **unsigned int**
ATUNCI următorul este convertit în **unsigned int**

Există încă un caz, special: dacă un element este **long** iar celălalt este **unsigned int** și dacă valoarea celui **unsigned int** nu poate fi reprezentată ca **long**, ambele elemente sunt convertite în **unsigned long**.

O dată aplicate aceste reguli de conversie, fiecare pereche de elemente este de același tip iar rezultatul fiecărei operații este de același tip cu cu cel al ambelor elemente.

De exemplu, să considerăm conversia de tip care apare în **Figura 2-2**. Pentru început, caracterul **ch** este convertit într-un **integer** iar **float f** este convertit în **double**. Apoi, rezultatul lui **ch/i** este convertit în **double** deoarece **f*d** este **double**. Rezultatul final este **double** deoarece, de data aceasta, ambele elemente sunt **double**.

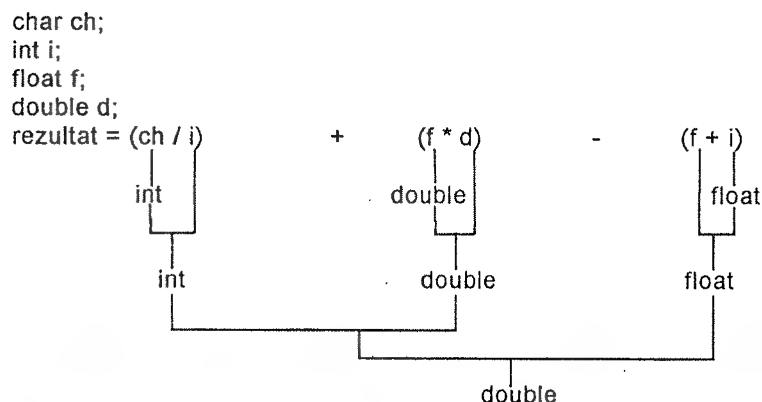


Figura 2-2. *Exemplu de conversie a tipurilor*

Modelatori

Puteți să forțați o expresie să devină de un anumit tip utilizând un *modelator*. Forma generală a unui modelator este:

(tip) expresie

unde *tip* este un tip de dată valid. De exemplu, pentru a vă asigura că expresia **x/2** este evaluată ca fiind de tipul **float**, veți scrie:

```
(float) x/2
```

Modelatorii sunt operatori tehnici. Ca operator, un modelator este unar și are aceeași prioritate ca oricare operator unar.

Deși modelatorii nu sunt uzuali în programare, ei pot fi foarte utili când acțiunea lor este necesară. De exemplu, să presupunem că doriți să folosiți un întreg pentru a controla o buclă, dar pentru a efectua calculul este necesară o parte fracționară, ca în următorul program.

```
#include <stdio.h>

void main(void) /* afiseaza i si i/2 cu zecimale */
```



```
{
    int i;

    for(i=1; i<=100; ++i)
        printf("%d / 2 este: %f\n", i, (float) i/2);
}
```

Fără modelatorul **(float)**, s-ar fi efectuat doar împărțire de întregi. Modelatorul asigură că va fi afișată partea fracționară a răspunsului.



NOTĂ: *Standardul propus pentru ANSI C++ a adăugat câțiva operatori noi de modelare, ca de exemplu **const_cast** și **static_cast**. Acești modelatori sunt discutați în Partea a doua.*

Spațieri și paranteze

Puteți să adăugați spații simple și de tabulare în expresiile în C pentru a le face mai ușor de citit. De exemplu, următoarele două expresii sunt identice.

```
x=10/y~(127/x);
x = 10 / y ~(127/x);
```

Parantezele în plus sau în exces nu implică erori sau încetinirea execuției expresiilor. Puteți să utilizați parantezele pentru a clarifica ordinea exactă a evaluării, atât pentru dvs. cât și pentru ceilalți. De exemplu, care dintre următoarele două expresii este mai ușor de citit?

```
x=y/2-34*temp&127;
x = (y/3) - ((34*temp) & 127);
```

Prescurtări în C

Există variațiuni ale instrucțiunilor de atribuire, uneori numite *prescurtări* în C, care simplifică codul pentru anumite operații de atribuire. De exemplu,

```
x = x+10;
```

poate fi scris ca:

```
x += 10;
```

Operatorul += spune compilatorului să atribuie lui **x** valoarea lui **x** plus 10.

Această prescurtare este operantă pentru toţi operatorii binari (aceia care solicită două elemente de operare). În general, instrucţiuni ca:

```
var = var operator expresie
```

pot fi scrise ca:

```
var operator = expresie
```

Ca alt exemplu,

```
x = x-100;
```

este acelaşi lucru cu:

```
x -= 100;
```



NOTĂ: *Notarea prescurtată este larg utilizată în scrierea programelor profesionale în C/C++; ar trebui să vă familiarizaţi cu ea.*

Capitolul 3

Instrucţiuni



Acest capitol prezintă instrucţiunile. În cel mai general sens, o instrucţiune este o porţiune a programului care poate fi executată. Aceasta înseamnă că o instrucţiune specifică o acţiune. Standardul ANSI C (şi cel propus pentru ANSI C++) împart instrucţiunile în următoarele grupe:

- Selecţie
- Iterare
- Salt
- Etichetă
- Expresie
- Bloc

Instrucţiunile de selecţie cuprind **if** şi **switch**. (Deseori este utilizat termenul *instrucţiune condiţională* în loc de „instrucţiune de selecţie".) Instrucţiunile de iterare sunt **while**, **for** şi **do-while**. Acestea mai sunt denumite *şi instrucţiuni de buclare*. Instrucţiunile de salt sunt **break**, **continue**, **goto** şi **return**. Instrucţiunile etichetă includ **case** şi **default** (discutate împreună cu instrucţiunea **switch**) şi etichetele (discutate cu **goto**). Instrucţiunile expresie sunt instrucţiuni compuse dintr-o expresie validă. Instrucţiunile bloc sunt simple blocuri de cod. (Amintiţi-vă că un bloc începe cu { şi se încheie cu }.) Standardul ANSI C++ propus mai denumeşte instrucţiunile bloc şi *instrucţiuni compuse*.

***NOTĂ:** C++ adaugă două tipuri de instrucţiuni: blocul **try** şi instrucţiunea de declarare. Ele sunt discutate în Partea a doua.*

De vreme ce multe instrucţiuni se bazează pe rezultatul unui test de condiţionare, să începem cu recapitularea conceptelor de adevărat şi de fals.

Adevărat şi Fals în C

Multe instrucţiuni în C se bazează pe o expresie de condiţionare care determină cursul acţiunilor următoare. O expresie condiţională este evaluată ca adevărat sau fals. În C, spre deosebire de alte limbaje, este adevărată orice valoare diferită de zero, inclusiv numerele negative. O valoare falsă este 0. Aceste concepte de adevărat şi fals permit o mare varietate de rutine care pot fi codate foarte eficient, aşa cum veţi vedea în curând.

***NOTĂ:** Chiar dacă propunerea de standard ANSI C++ defineşte un tip de dată Boolean numit **bool** (care poate să aibă doar valorile **adevărat** şi **fals**), C++ păstrează aceleaşi concepte generale de adevărat şi fals ca şi C.*

Instrucţiuni de selecţie

C admite două tipuri de instrucţiuni de selecţie: **if** şi **switch**. În plus, operatorul ? este în anumite condiţii o alternativă a lui **if**.

if

Forma generală a instrucţiunii **if** este:

if (*expresie) instrucţiune;*
else *instrucţiune*;

unde *instrucţiune* poate să fie o singură instrucţiune, un bloc de instrucţiuni sau nici una (în cazul instrucţiunilor vide). Clauza **else** este opţională.

Dacă *expresie* este evaluată ca adevărat (orice altceva în afară de 0), atunci este executată instrucţiunea sau blocul care formează obiectul lui **if**; altfel, este executată instrucţiunea sau blocul care face obiectul lui **else**, dacă există. Reţineţi că se va executa ori codul asociat lui **if** ori cel asociat lui **else**, niciodată amândouă.

Instrucţiunea de condiţionare care controlează pe **if** trebuie să determine un rezultat scalar. Un *scalar* este unul din tipurile întreg, caracter, pointer sau în virgulă mobilă. Totuşi, rareori se utilizează un număr în virgulă mobilă pentru a controla o instrucţiune de condiţionare deoarece acesta încetineşte considerabil execuţia. (Sunt necesare mai multe instrucţiuni pentru a efectua o operaţie în virgulă mobilă decât pentru a executa operaţii cu întregi sau cu caractere.)

Următorul program conţine un exemplu de utilizare a lui **if**. Programul redă o versiune simplă a jocului „ghiceşte numărul magic". El afişează mesajul ****Corect**** atunci când jucătorul ghiceşte numărul magic. Numărul magic este obţinut prin utilizarea generatorului de numere aleatorii din C **rand()**, care returnează un număr arbitrar între 0 şi **RAND_MAX** (valoare întreagă egală sau chiar mai mare decât 32.767). **rand()** cere fişierul antet STDLIB.H.

```
/* Program #1 pentru numarul magic. */
#include <stdio.h>
#include <stdlib.h>

void main(void)
{
   int magic;  /* numar magic */
   int ghici;  /* numar jucator */

   magic = rand();   /* genereaza numarul magic */
   printf("ghiceste numarul magic: ");
```

```
        scanf("%d", &ghici);
        if(ghici == magic)  printf("**Corect**");
    }
```

Continuând programul numărului magic, următoarea versiune ilustrează utilizarea instrucțiunii **else** pentru a afișa un mesaj ca răspuns la introducerea unui număr greșit.

```
/* Program #2 pentru numarul magic. */
#include <stdio.h>
#include <stdlib.h>

void main(void)
{
    int magic;  /* numar magic */
    int ghici;  /* numar jucator */

    magic = rand();   /* genereaza numarul magic */
    printf("ghiceste numarul magic: ");
    scanf("%d", &ghici);
    if(ghici == magic)  printf("**Corect**");
    else printf("Gresit");
}
```

if imbricat

Un **if** imbricat este un **if** care este obiectul unui alt **if** sau al unui **else**. În programare **if** imbricat este foarte uzual. Într-un **if** imbricat, o instrucțiune **else** se referă întotdeauna la cea mai apropiată instrucțiune care se află în același bloc cu **else** și care nu este deja asociată unui alt **else**. De exemplu,

```
if(i)
{
    if(j) instructiune 1;
    if(k) instructiune 2;   /* acest if */
    else  instructiune 3;   /* este asociat acestui else */
}
else      instructiune 4;   /* asociat cu if(i) */
```

Așa cum am explicat, **else** din final nu este asociat cu **if(j)** deoarece el nu se găsește în același bloc. El este asociat lui **if(i)**. De asemenea, **else** din interior este asociat lui **if(k)**, deoarece acesta este cel mai apropiat **if**.

Standardul ANSI C specifică un număr de cel puțin 15 niveluri de imbricare. În practică, majoritatea compilatoarelor permit mult mai multe. Mai important este faptul că propunerea de ANSI C++ sugerează că într-un program în C++ trebuie să fie permise cel puțin 256 de niveluri de imbricare pentru **if**. Totuși, imbricarea peste mai mult de câteva niveluri este rareori necesară, iar imbricarea excesivă îngreunează înțelegerea algoritmului.

Puteți să utilizați un **if** imbricat pentru îmbunătățirea programului numărului magic oferind jucătorului o apreciere asupra numărului greșit.

```
/* Program #3 pentru numarul magic. */
#include <stdio.h>
#include <stdlib.h>

void main(void)
{
    int magic;  /* numar magic */
    int ghici;  /* numar jucator */

    magic = rand();   /* genereaza un numar aleator */

    printf("ghiceste numarul magic: ");
    scanf("%d", &ghici);
    if(ghici == magic)  {
      printf("**Corect**");
      printf(" %d este numarul magic\n");
    }
    else {
       printf("Gresit, ");
       if(ghici > magic)  printf("prea mare\n");
       else printf("prea mic\n");
    }
}
```

Scara *if-else-if*

O construcție de programare uzuală este *scara if-else-if*, denumită astfel datorită felului în care se prezintă. Forma sa generală este:

if (*expresie*) *instrucțiune*;
else
    if (*expresie*) *instrucțiune;*
    else

if (*expresie*) *instrucțiune*
.
.
.
else *instrucțiune;*

Condițiile sunt evaluate de sus în jos. Imediat ce este întâlnită o condiție adevărată, va fi executată instrucțiunea asociată iar restul scării va fi ignorat. Dacă nici una dintre condiții nu este adevărată, va fi executat **else** final. Aceasta înseamnă că dacă toate celelalte condiții cad, va fi executată ultima instrucțiune **else**. Dacă nu există un **else** final, nu va avea loc nici o acțiune în cazul în care celelalte condiții sunt false.

Chiar dacă modul de prezentare în trepte a scării precedente if-else-if este tehnic corect, el poate să ducă la o exagerare a numărului de identări. Din acest motiv, scara if-else-if este de obicei prezentată astfel:

if (*expresie*)
    instrucțiune;
else if (*expresie*)
    instrucțiune;
else if (*expresie*)
    instrucțiune;
.
.
.
else
    instrucțiune;

Utilizând o scară if-else-if, programul cu numere magice devine:

```
/* Program #4 pentru numarul magic. */
#include <stdio.h>
#include <stdlib.h>

void main(void)
{
    int magic;  /* numar magic */
    int ghici;  /* numar jucator */

    magic = rand();   /* genereaza numarul magic */

    printf("ghiceste numarul magic: ");
    scanf("%d", &ghici);
```



```
    if(ghici == magic)  {
      printf("**Corect**");
      printf(" %d este numarul magic", magic);
    }
    else if(ghici > magic)
       printf("Gresit, prea mare");
    else   printf("Gresit, prea mic");
}
```

Alternativa ?

Puteți să utilizați operatorul **?** pentru a înlocui instrucțiunile **if-else** de forma următoare:

if (*condiție*) *expresie;*
else *expresie;*

Dar subiectul, atât pentru **if**, cât și pentru **else**, trebuie să fie o singură expresie - nu o altă instrucțiune.

Operatorul **?** este numit *operator ternar* deoarece el cere trei elemente asupra cărora operează. El are forma generală:

Exp1 ? *Exp2* : *Exp3*

unde *Exp1*, *Exp2* și *Exp3* sunt expresii. Rețineți utilizarea și plasarea celor două puncte.

Valoarea unei expresii **?** este evaluată astfel: se evaluează *Exp1*. Dacă este adevărată, *Exp2* este evaluată și devine valoarea întregii expresii ?. Dacă *Exp1* este falsă, atunci se evaluează *Exp3* și valoarea ei devine valoarea expresiei ?. De exemplu să considerăm:

```
x = 10;
y = x>9 ? 100 : 200;
```

În acest exemplu, lui **y** îi este atribuită valoarea 100. Dacă **x** ar fi fost mai mic decât 9, **y** ar fi primit valoarea 200. Același cod scris cu instrucțiuni **if-else** ar fi:

```
x = 10;
if(x>9) y = 100;
else y = 200;
```

Următorul program utilizează operatorul **?** pentru a ridica la pătrat o valoare întreagă introdusă de către utilizator. Program păstrează semnul (10 la pătrat este 100 iar -10 la pătrat este -100).

```
#include <stdio.h>

void main(void)
{
    int ipatrat, i;

    printf("Introduceti un numar: ");
    scanf("%d", &i);

    ipatrat = i>0 ? i*i : -(i*i);
    printf("%d patrat este %d", i, ipatrat);
}
```

Utilizarea operatorului **?** pentru înlocuirea secvenței **if-else** nu este restrânsă doar la instrucțiunile de atribuire. Amintiți-vă că toate funcțiile (cu excepția celor declarate **void**) pot să returneze o valoare. Astfel, puteți să utilizați una sau mai multe apelări ale unei funcții într-o expresie. Când este întâlnit numele funcției, ea este executată astfel încât să fie determinată valoarea returnată de ea. De aceea, puteți să executați una sau mai multe apeluri de funcție utilizând operatorul **?** plasând apelurile în expresiile care îi servesc drept operanzi, ca mai jos:

```
#include <stdio.h.

int f1(int n);
int f2(void);

void main(void)
{
    int t;
    printf("Introducati un numar: ");
    scanf("%d", &t);

    /* afiseaza mesajul corespunzator */
    t ? f1(t) + f2() : printf("s-a introdus 0");
}
f1(int n)
{
    printf("%d ", n);
    return 0;
}
f2(void)
{
    printf("introdus");
    return 0;
}
```


Introducerea lui 0 în acest exemplu apelează funcția **printf()** și afișează mesajul **s-a introdus 0**.

Dacă introduceți orice alt număr, se vor executa atât **f1()** cât și **f2()**. Rețineți că în acest exemplu se renunță la valoarea expresiei **?**. Nu trebuie să îi atribuiți nimic.



ATENȚIE: *Unele compilatoare de C/C++ rearanjează ordinea de evaluare a unei expresii în încercarea de a optimiza codul obiect. Aceasta poate să determine ca funcțiile care formează expresia supusă operatorului ? să se execute într-o ordine nedorită.*

Utilizând operatorul **?**, puteți să rescrieți programul cu numere magice.

```
/* Program #5 pentru numarul magic. */
#include <stdio.h>
#include <stdlib.h>

void main(void)
{
    int magic;  /* numar magic */
    int ghici;  /* numar jucator */

    magic = rand();   /* genereaza numarul magic */

    printf("ghiceste numarul magic: ");
    scanf("%d", &ghici);

    if(ghici == magic)  {
      printf("**Corect**");
      printf(" %d este numarul magic", magic);
    }
    else
          ghici > magic ? printf("Prea mare") :
                          printf("Prea mic");
}
```

Aici, operatorul **?** afișează mesajul corect în funcție de testul **ghici>magic**.

Expresii de condiţionare

Uneori începătorii în C (şi C++) sunt derutaţi de faptul că se poate folosi orice expresie validă pentru controlul asupra operatorilor **if** sau **?**. Aceasta înseamnă că nu sunteţi limitaţi la expresii care implică operatorii relaţionali şi logici (precum în cazul limbajelor BASIC sau Pascal). Expresia este evaluată simplu ca valoare zero sau non-zero. De exemplu, următorul program citeşte doi întregi introduşi de la tastatură şi afişează câtul lor. El foloseşte o instrucţiune **if**, controlată de al doilea număr, pentru a evita eroarea obţinută prin împărţirea la zero.

```
/* Imparte primul numar la al doilea. */
#include <stdio.h>

void main(void)
{
    int a, b;
    printf("Introduceti doua numere: ");
    scanf("%d%d", &a, &b);

    if(b) printf("%d\n", a/b);
    else printf("Nu pot imparti la zero.\n");
}
```

Programul funcţionează deoarece dacă **if** este 0, condiţia care îl controlează pe **if** este falsă şi se va executa **else**. Altfel, condiţia este adevărată (non-zero) şi are loc împărţirea. Scrierea instrucţiunii **if** astfel:

```
if(b != 0) printf("%d\n", a/b);
```

este inutilă, potenţial ineficientă şi considerată ca lipsită de stil.

switch

C are o instrucţiune de condiţionare multiramură, numită **switch**, care testează succesiv valoarea unei expresii faţă de o listă de constante de tip caracter sau întreg. Când se întâlneşte o coincidenţă, se execută instrucţiunea asociată acelei constante. Forma generală a instrucţiunii **switch** este :

switch (*expresie*) {
    case *constanta1*:
        secvenţă de instrucţiuni
        break;
    case *constanta2:*
        secvenţă de instrucţiuni
        break;
    case *constanta3:*
        secvenţa de instrucţiuni
        break:
    .
    .
    .
    default
        secvenţă de instrucţiuni
}

Este testată valoarea din *expresie*, faţă de valorile constantelor specificate în instrucţiunile **case**. Când se întâlneşte o coincidenţă, se execută secvenţa de instrucţiuni asociată acelui **case** pînă la instrucţiunea **break** sau până când se ajunge la finalul instrucţiunii **switch**. Instrucţiunea **default** se execută dacă nu este întâlnită nici o coincidenţă. **default** este opţional şi, dacă nu este prezent, nu are loc nici o acţiune dacă nu se găseşte nici o potrivire.

Standardul ANSI C stipulează că **switch** poate să aibă cel puţin 257 de instrucţiuni de tip **case**. Standardul propus pentru ANSI C++ recomandă să poată introduce *cel puţin* 16.384 de instrucţiuni de tip **case**. În practică, din motive de eficienţă, veţi dori să limitaţi numărul de instrucţiuni **case** la o valoare mai mică. Deşi **case** este o instrucţiune etichetă, ea nu poate exista de una singură, în afara unei instrucţiuni **switch**.

Instrucţiunea **break** este o instrucţiune de salt în C. Puteţi să o utilizaţi la fel de bine în bucle, ca şi în instrucţiuni **switch** (după cum se vede în secţiunea „Instrucţiuni de iterare"). Când se întâlneşte **break** într-o construcţie **switch**, programul execută un salt la linia de cod care urmează instrucţiunii **switch**.

Trebuie să ştiţi trei lucruri importante despre instrucţiunea **switch**:

- **switch** diferă de **if** prin aceea că testează doar egalitatea, în timp ce **if** poate să evalueze orice tip de expresie relaţională sau logică.
- În acelaşi **switch** nu pot exista două constante **case** cu valori identice. Desigur, două instrucţiuni **switch**, una inclusă în cealaltă, pot să aibă aceeaşi constantă **case**.
- Dacă în instrucţiunea **switch** sunt utilizate constante de tip caracter, ele sunt automat convertite în întregi.

Instrucţiunea **switch** este deseori folosită pentru a prelucra comenzile de la tastatură, cum ar fi selecţia dintr-un meniu. Aşa cum se arată în continuare, funcţia **menu()** afişează un meniu pentru un program de verificare ortografică şi care apelează procedura corespunzătoare:

```
void menu(void)
{
    char ch;

    printf("1. Control ortografic\n");
    printf("2. Corecteaza erorile de ortografie\n");
    printf("3. Afiseaza erorile de ortografie\n");
    printf("   Apasati orice tasta pentru a iesi din
           program\n");
    printf("   Introduceti o optiune: ");

    ch = getchar();  /* citeste selectia de la tastatura */
    switch(ch) {
          case '1':
                control_orto();
                break;
          case '2':
                corect_erori();
                break;
          case '3':
                afis_erori();
                break;
          default :
                printf("Nu s-a selectat nici o optiune");
    }
}
```

Practic, instrucţiunile **break** sunt opţionale în construcţiile **switch**. Ele încheie secvenţa instrucţiunilor asociate unei constante. Dacă sunt omise, execuţia va continua cu instrucţiunile din următorul **case** până este întâlnit un **break** sau până se ajunge la sfârşitul lui **switch**. De exemplu, funcţia următoare foloseşte facilitatea de „trecere liberă" prin **case** pentru a simplifica codul unui gestionar al intrărilor de la un driver de dispozitiv:

```
/* Proceseaza o valoare */
void manev_intr(int i)
{
    int marcaj;

    marcaj = -1;

    switch(i)  {
          case 1:  /* aceste case au secvente */
          case 2:  /* comune de instructiuni */
          case 3:
                marcaj = 0;
                break;
          case 4:
                marcaj = 1;
          case 5:
                eroare(marcaj);
                break;
          default:
                procesat(i);
    }
}
```


Acest exemplu ilustrează două aspecte ale lui **switch**. Primul, puteţi avea instrucţiuni **case** care nu au asociate secvenţe de instrucţiuni. Când apar, pur şi simplu execuţia sare la următorul **case**. În acest exemplu, primele trei **case** execută aceleaşi instrucţiuni, care sunt:

```
marcaj = 0;
break;
```

În al doilea rând, execuţia unei secvenţe de instrucţiuni continuă cu următorul **case** dacă nu este prezentă instrucţiunea **break**. Dacă **i** este 4, **marcaj** este 1 şi, deoarece nu există nici o instrucţiune **break** la sfârşitul acestui **case**, execuţia continuă şi se execută apelarea lui **eroare(marcaj)**. Dacă **i** ar fi fost 5, **eroare(marcaj)** ar fi trebuit să fie apelat cu valoarea -1 (în loc de 1).

Faptul că nişte **case** pot rula împreună atunci când nu este prezent nici un **break** previne repetarea inutilă a instrucţiunilor, rezultând un cod mult mai eficient.

Instrucţiuni *switch* imbricate

Puteţi să aveţi un **switch** inclus într-o secvenţă de instrucţiuni a unui alt **switch**, exterior. Chiar dacă unele constante **case** din **switch** interior şi din cel exterior conţin valori comune, nu apar conflicte. De exemplu, următorul fragment de cod este perfect valabil:

```
switch(x)  {
    case 1:
          switch(y)  {
                case 0: printf("impartire la 0, eroare");
                      break;
                case 1: procesat(x,y);
```

```
        }
        break;
    case 2:
    .
    .
    .
```

Instrucțiuni de iterare

În C, ca și în alte limbajele de programare moderne, instrucțiunile de iterare (denumite și *bucle*) permit ca un set de instrucțiuni să se execute repetat, până se îndeplinește o anumită condiție. Această condiție poate fi predefinită (ca în buclele **for**) sau cu sfârșit deschis (ca în buclele **while** și **do-while**).

Bucla *for*

Conceptul general al buclei **for** în C este reflectat într-o formă sau alta în toate procedurile limbajele structurate de programare. Totuși, în C, el capătă flexibilitate și putere neașteptată.

Forma generală a instrucțiunii **for** este:

for (*inițializare; condiție; increment) instrucțiune;*

Bucla **for** permite multe variații. Totuși, *inițializare* este în general o instrucțiune de atribuire utilizată pentru a inițializa variabila de control a buclei. *Condiție* este o expresie relațională care determină ieșirea din buclă. *Increment* definește modul în care se modifică variabila de control a buclei de fiecare dată când aceasta se repetă. Trebuie să separați aceste trei secțiuni principale prin punct și virgulă. Bucla **for** continuă să se execute atâta timp cât condiția este adevărată. Când condiția devine falsă, execuția programului se reia de la instrucțiunea care urmează lui **for**.

În următorul program, o buclă **for** este utilizată pentru a afișa pe ecran numerele de la 1 la 100:

```
#include <stdio.h>

void main(void)
{
    int x;

    for(x=1; x <= 100; x++) printf("%d ", x);
}
```


În buclă, **x** este inițializat cu 1 și apoi comparat cu 100. De vreme ce **x** este mai mic decât 100, se apelează **printf()** iar bucla se reia, ceea ce face ca **x** să crească cu 1 și să fie testat din nou pentru a se vedea dacă încă este mai mic sau egal cu 100. Dacă este așa, se apelează **printf()**. Acest proces se repetă până când **x** devine mai mare ca 100, moment în care bucla se încheie. În acest exemplu, **x** este variabila de control, care este modificată și verificată de fiecare dată când bucla se repetă.

Următorul exemplu este o buclă **for** care reia instrucțiuni multiple:

```
for(x=100; x != 65; x -= 5)  {
    z =x*x;
    printf("Patratul lui %d, %f", x, z);
}
```

Atât ridicarea la pătrat a lui **x** cât și apelarea lui **printf()** sunt executate până când **x** devine egal cu 65. Rețineți că bucla este *rulată în sens invers*: **x** este inițializat cu 100 și la fiecare repetare a buclei se scad din el câte 5 unități.

În bucla **for** testul de condiționare este efectuat la începutul ciclului. Aceasta înseamnă că un cod din interiorul buclei poate să nu fie executat de loc dacă este falsă condiția de început. De exemplu, în

```
x = 10;
for(y=10; y!=x; ++y) printf("%d", y);
printf("%d", y);   /* aceasta este singura instructiune
                      printf care se executa */
```

bucla nu se va executa niciodată deoarece **x** și **y** sunt egale în momentul intrării în buclă. Deoarece aceasta determină evaluarea expresiei de condiționare ca falsă, nici corpul buclei, nici zona de incrementare a buclei nu se execută. Astfel, **y** are în continuare valoarea 10 și singura ieșire produsă este numărul 10 afișat o singură dată pe ecran.

Versiuni ale buclei *for*

Secțiunea anterioară a prezentat cea mai uzuală formă a buclei **for**. Însă, pentru a-i mări puterea, flexibilitatea și aplicabilitatea în situații specifice de programare, sunt permise mai multe variațiuni ale lui **for**.

Una dintre cele mai folosite variațiuni folosește operatorul *virgulă* pentru a permite ca bucla să fie controlată de două sau mai multe variabile. (Amintiți-vă că operatorul *virgulă* se utilizează pentru a alătura un număr de expresii în modul „fă asta și asta", după cum s-a arătat în **Capitolul 2**.) De exemplu, variabilele **x** și **y** controlează următoarea buclă și amândouă sunt inițializate în interiorul instrucțiunii **for**:

```
for(x=0, y=0; x+y<10; ++x)  {
    y = getchar();
    y = y-'0'; /* scade codul ASCII pentru 0 din y */
    .
    .
    .
}
```

Virgula separă cele două instrucțiuni de inițializare. De fiecare dată când se repetă bucla, **x** este incrementat iar valoarea lui **y** este introdusă de la tastatură. Atât **x** cât şi **y** trebuie să aibă o anumită valoare pentru ca bucla să se încheie. Chiar dacă valoarea lui **y** este introdusă de la tastatură, **y** trebuie să fie inițializat cu 0 astfel încât valoarea sa să fie stabilită înainte de prima evaluare a expresiei de condiționare. (Dacă **y** nu ar fi inițializat, ar putea întâmplător să conțină valoarea 10, condiția de test devenind falsă şi împiedicând execuția buclei.)

Funcția **converg()** din exemplul următor, prezintă o buclă cu mai multe variabile de control. Ea afişează un şir tipărind caracterele de la ambele capete, convergând către mijlocul liniei specificate. Aceasta solicită poziționarea cursorului în diverse puncte distincte pe ecran. Deoarece C/C++ rulează într-o varietate mare de medii, ele nu definesc o funcție de poziționare a cursorului. Totuşi, virtual, toate compilatoarele de C/C++ asigură una, chiar dacă numele ei poate să difere. Următorul program foloseşte funcția de poziționare a cursorului din Borland, care se numeşte **gotoxy()**. (Ea solicită fişierul antet CONIO.H.)

```
/* Versiunea Borland. */
#include <stdio.h>
#include <conio.h>
#include <string.h>

void converg(int line, char *mesaj);

void main (void)
{
    converg(10, "Acesta este un test pentru converg().");
}
/* Aceasta functie afiseaza un sir incepind de la stinga
   liniei specificate. Ea scrie caracterele de la ambele
   capete, convergind catre mijloc. */
void converg(int linie, char *mesaj)
{
    int i, j;
    for(i=1, j=strlen(mesaj);  i<j;  i++,  j--)  {
            gotoxy(i, line); printf("%c", mesaj[i-1]);
            gotoxy(j, line); printf("%c", mesaj[j-1]);
    }
}
```


Echivalentul Microsoft al funcției **gotoxy()** este **_settextposition()**, care foloseşte fişierul antet **graph.h**. Programul de mai sus, codificat pentru Microsoft C/C++, devine:

```
/* Versiune Microsoft. */
#include <stdio.h>
#include <conio.h>
#include <string.h>

void converg(int line, char *mesaj);

void main (void)
{
    converg(10, "Acesta este un test pentru converg().");
}
/* Aceasta functie afiseaza un sir incepind de la stinga
   liniei specificate. Ea scrie caracterele de la ambele
   capete, convergind catre mijloc. */
void converg(int linie, char *mesaj)
{
    int i, j;
    for(i=1, j=strlen(mesaj);  i<j;  i++,  j--)  {
            _settextposition(linie, i);
            printf("%c", mesaj[i-1]);
            _settextposition(linie, j);
            print("%c", mesaj[j-1];
    }
}
```

Dacă folosiți alt compilator de C/C++, va fi necesar să consultați manualul utilizatorului pentru a afla numele funcției de poziționare a cursorului.

În ambele versiuni ale lui **converg()**, bucla **for** utilizează două variabile de control, **i** şi **j**, pentru a avea acces la şir pe la ambele capete. Pe măsură ce se reia bucla, **i** creşte, iar **j** scade. Bucla se încheie când **i** este mai mare sau egal cu **j**, aceasta asigurând că toate caracterele sunt scrise.

Expresia de condiționare nu implică testarea variabilei de control a buclei față de o anumită valoare. De fapt, condiția poate fi orice instrucțiune relațională sau logică. Aceasta înseamnă că puteți să testați multe condiții de încheiere.

De exemplu, puteți să utilizați următoarea funcție pentru a introduce un utilizator într-un sistem la distanță. Utilizatorul poate încerca de trei ori să introducă parola. Bucla se încheie când au fost epuizate cele trei încercări sau când utilizatorul introduce parola corectă.

```
void semn(void)
{
    char sir[20];
    int x;

    for(x=0; x<3 && strcmp(sir, "parola"); ++x) {

        printf("Va rog, introduce-ti parola:");
        gets(sir);
    }
if(x==3) return;
/* altfel conectati utilizatorul la... */
}
```

Această funcție folosește **strcmp()**, funcția de bibliotecă standard care compară două șiruri și returnează 0 dacă ele sunt identice.

Amintiți-vă că fiecare dintre cele trei secțiuni ale buclei **for** poate să conțină orice expresie validă. Nu este necesar ca expresiile să aibă vreo legătură cu scopul utilizării secțiunilor. Ținând minte aceasta, să luăm următorul exemplu:

```
#include <stdio.h>

int numpatrat(int num);
int numcitit(void);
int prompt(void);

void main(void)
{
    int t;
    for(prompt(); t=numcitit(); prompt())
        numpatrat(t);
}

prompt(void)
{
    printf("Introduceti un numar: ");
    return 0;
}
```



```
numcitit(void)
{
    int t;

    scanf("%d", &t);
    return t;
}

numpatrat(int num)
{
    printf("%d", num*num);
    return num*num;
}
```

Să privim cu atenție la bucla **for** din **main()**. Remarcați că fiecare parte a buclei **for** este compusă din apelarea unor funcții care solicită utilizatorului și citesc un număr introdus de la tastatură. Dacă numărul introdus este 0, bucla se încheie deoarece expresia de condiționare va fi falsă. Altfel, numărul va fi ridicat la pătrat. Pentru aceasta, bucla **for** folosește secvențele de inițializare și de incrementare într-un mod neobișnuit, dar perfect valabil.

O altă trăsătură interesantă a buclei **for** este aceea că pot lipsi părți din definiția sa generală. De fapt, nu este necesar să existe nici o expresie pentru nici una dintre secțiuni - expresiile sunt opționale. De exemplu, această buclă va rula până când utilizatorul va introduce **123**:

```
for(x=o; x!=123; ) scanf("%d", &x);
```

Remarcați că zona de incrementare a definiției **for** este liberă. Aceasta înseamnă că de fiecare dată când se repetă bucla, este testat **x** pentru a vedea dacă este egal cu 123, dar nici o altă acțiune nu are loc. Dacă însă scrieți **123** de la tastatură condiția devine falsă și bucla se încheie.

De multe ori inițializarea se întâlnește în afara instrucțiunii **for**. Aceasta se întâmplă deseori când starea inițială a variabilei de control a buclei trebuie să fie calculată prin metode mai complexe, ca în acest exemplu:

```
gets(s);  /* citeste un sir in s */
if(*s) x = strlen(s);  /* preia lungimea sirului */
else x = 10;

for( ; x<10; ) {
    printf("%d", x);
    ++x;
}
```

Secțiunea de inițializare a fost lăsată liberă, iar x capătă valoare înainte de intrarea în buclă.

Bucla infinită

Chiar dacă puteți utiliza orice instrucțiune de buclare pentru a crea o buclă infinită, în mod clasic este utilizat în acest scop **for**. De vreme ce nu este obligatorie prezența nici uneia din cele trei expresii care formează bucla **for**, puteți să creați o buclă fără sfârșit lăsând necompletată condiția de control, așa cum se arată aici:

```
for( ; ; ) printf(" Aceasta bucla va rula la infinit.\n");
```

Când expresia de condiționare este absentă, se presupune că este adevărată. Puteți să aveți o expresie de inițializare și de incrementare, dar programatorii în C/C++ utilizează mai frecvent construcția **for(;;)** pentru a semnala o buclă infinită.

De fapt, construcția **for(;;)** nu garantează o buclă infinită deoarece instrucțiunea **break**, întâlnită oriunde în interiorul corpului buclei, determină încheierea imediată a acesteia (**break** va fi discutată mai târziu în acest capitol). Programul se reia atunci de la codul care urmează buclei, așa cum este prezentat mai jos:

```
ch = '\0';

for( ; ; ) {
    ch = getchar(); /* preia un caracter */
    if(ch=='A') break; /* iese din bucla */
}

printf("ati scris un A");
```

Această buclă va rula până când utilizatorul va scrie un A de la tastatură.

Bucle *for* fără corp

O instrucțiune poate să fie goală. Aceasta înseamnă că și corpul buclei **for** (sau al oricărei alte bucle) poate, de asemenea, să fie gol. Puteți folosi acest lucru pentru a mări eficiența anumitor algoritmi și pentru a crea bucle de întârziere.

O sarcină uzuală de programare este cea de îndepărtare a spațiilor dintr-un stream de intrare. De exemplu, un program de baze de date poate să permită o cerere de genul „prezintă toate bilanțurile mai mici de 400". Baza de date necesită introducerea distinctă a fiecărui cuvânt, fără spații, adică ea recunoaște „**baza**" dar nu „ **baza**". Următoarea buclă îndepărtează spațiile libere de început din streamul indicat de către **sir**.

```
for( ; *str == ' '; str++) ;
```

După cum puteți vedea, această buclă nu are corp - și nici nu ar avea nevoie.

Buclele de *întârziere* sunt deseori folosite în programe. Următorul cod arată cum se creează una prin utilizarea lui **for**:

```
for(t=0; t<NISTE_VALORI; t++) ;
```

Bucla *while*

A doua buclă disponibilă în C este bucla **while**. Forma sa generală este:

while (*condiție*) *instrucțiune*;

unde *instrucțiune* este o instrucțiune vidă, o singură instrucțiune sau un bloc de instrucțiuni. *Condiție* poate să fie orice expresie; este adevărată pentru orice valoare non-zero. Bucla se reia atât timp cât condiția este adevărată. Când condiția devine falsă, controlul programului trece la linia de cod imediat următoare buclei.

Următorul exemplu arată o rutină de introducere de la tastatură care se reia până când utilizatorul introduce **A**:

```
asteapta_caracter(void)
{
    char ch;

    ch = '\0'; /* initializeaza ch */
    while(ch != 'A') ch = getchar();
    return ch;
}
```

La început, **ch** este inițializat cu 0. Ca variabilă locală, valoarea sa nu este cunoscută când se execută **asteapta_caracter**. Apoi bucla **while** verifică dacă **ch** este egal cu **A**. Deoarece **ch** este inițializat cu 0, testul este adevărat, iar bucla începe execuția. De fiecare dată când apăsați o tastă, se testează din nou condiția. Când introduceți **A**, condiția devine falsă deoarece **ch** este egal cu **A**, iar bucla se încheie.

Ca și bucla **for**, bucla **while** verifică condiția de testare la începutul buclei, ceea ce înseamnă că nu se va executa corpul buclei dacă acea condiție este falsă. Acest fapt poate să elimine necesitatea de a executa o altă condiție de testare înainte de intrarea în buclă, după cum se arată în funcția **umple()**. Ea adaugă

spaţii la sfârşitul unui şir pentru a-l face de o lungime predefinită. Dacă şirul are deja lungimea dorită, nu se vor mai adăuga spaţii.

```
#include <stdio.h>
#include <string.h>

void umple(char *s, int lungime);

void main(void)
{
    char sir[80];

    strcpy(sir, "acesta este un test");
    umple(sir, 40);
    printf("%d", strlen(sir));
}

/* Adauga spatii la sfirsitul unui sir. */
void umple(char *s, int lungime)
{
    int l;

    l = strlen(s); /* determina lungimea sa */

    while(l<lungime) {
          s[l] = ' ';  /* introduce un spatiu */
          l++;
    }
    s[l] = '\0';  /* sirul trebuie sa se încheie cu null */
}
```

Cele două argumente ale lui **umple()** sunt **s**, un pointer la un şir supus măririi, şi **lungime**, numărul de caractere pe care trebuie să îl aibă **s**. Dacă lungimea şirului **s** este deja egală sau mai mare decât **lungime**, codul din interiorul buclei **while** nu se execută. Dacă **s** este mai scurt decât **lungime**, **umple()** adaugă numărul necesar de spaţii. Funcţia **strlen()**, parte a bibliotecii standard, returnează lungimea şirului.

Dacă sunt necesare mai multe condiţii independente pentru a încheia bucla **while**, se creează o valoare comună care să le sintetizeze în expresia de condiţionare. Valoarea acestei variabile este stabilită în diverse puncte din cadrul buclei. În acest exemplu,


```
void func1(void)
{
    int lucreaza;

    lucreaza = 1; /* adica, adevarat */

    while(lucreaza) {
          lucreaza = prelucr1();
          if(lucreaza)
                lucreaza = prelucr2();
          if(lucreaza)
                lucreaza = prelucr3();
    }
}
```

oricare din cele trei rutine poate să returneze fals şi să determine ieşirea din buclă.

Nu este necesar să existe vreo instrucţiune în corpul buclei **while**. De exemplu,

```
while((ch=getchar()) != 'A');
```

va rula până când utilizatorul scrie A. Dacă vă este greu să introduceţi o atribuire în interiorul expresiei de condiţionare **while** amintiţi-vă că semnul egal este doar un operator care evaluează expresia la valoarea membrului drept.

Bucla *do-while*

Spre deosebire de buclele **for** şi **while**, care testează condiţia din buclă la începutul său, bucla **do-while** o verifică la sfârşit. Aceasta înseamnă că bucla **do-while** se execută cel puţin o dată. Forma generală a buclei **do-while** este:

do {
  instrucţiune;
} while(*condiţie*);

Chiar dacă acoladele nu sunt necesare atunci când este prezentă o singură instrucţiune, ele se folosesc de obicei pentru a evita confuziile (ale dvs., nu ale compilatorului) cu **while**. Bucla **do-while** se iterează până când *condiţie* devine falsă.

Următoarea buclă **do-while** va citi numere introduse de la tastatură până va găsi un număr mai mic sau egal cu 100.

```
do {
    scanf("%d", &num);
} while(num > 100);
```

Probabil că cea mai obişnuită utilizare a buclei **do-while** este într-o funcţie de selectare dintr-un meniu. Când utilizatorul introduce un răspuns valid, acesta este returnat ca valoare a funcţiei. Un răspuns incorect va determina o nouă solicitare. Următorul cod prezintă o variantă îmbunătăţită a meniului de cercetare-verificare prezentat mai devreme în acest capitol:

```
void menu(void)
{
    char ch;

    printf("1. Control ortografic\n");
    printf("2. Corecteaza erorile de ortografie\n");
    printf("3. Afiseaza erorile de ortografie\n");

    printf("   Introduceti o optiune: ");

    do {
      ch = getchar();  /* citeste selectia de la tastatura */
      switch(ch) {
          case '1':
                control_orto();
                break;
          case '2':
                corect_erori();
                break;
          case '3':
                afis_erori();
                break;
      }
    } while(ch!='1' && ch!= '2' && ch!='3');
}
```

Aici, bucla **do-while** este o alegere bună deoarece veţi dori ca un meniu să se execute cel puţin o dată. După ce opţiunile au fost afişate, programul se va relua până când se va selecta o opţiune validă.

Instrucţiuni de salt

C are patru instrucţiuni care execută ramificări necondiţionate: **return, goto, break** şi **continue**. Dintre acestea, **return** şi **goto** pot să se găsească oriunde în program. Instrucţiunile **break** şi **continue** pot fi utilizate împreună cu oricare din instrucţiunile de buclare. Aşa cum s-a discutat anterior, în acest capitol, puteţi să mai utilizaţi **break** şi împreună cu **switch**.

Instrucţiunea *return*

Instrucţiunea **return** este utilizată pentru reîntoarcerea dintr-o funcţie. E caracterizată drept instrucţiune de salt deoarece determină execuţia să se reîntoarcă (să sară înapoi) în punctul în care a fost apelată funcţia. **return** poate sau nu să aibă asociată o valoare. Dacă **return** are o astfel de valoare, aceasta devine valoarea returnată de funcţie. În C, o funcţie care nu este **void** nu trebuie neapărat să returneze o valoare. Dacă nu se specifică nici o valoare, se va returna ceva, la întâmplare. Totuşi, în C++, o funcţie care nu este **void** *trebuie* să returneze o valoare. Aceasta înseamnă că în C++, dacă se menţionează că o funcţie returnează o valoare, orice instrucţiune **return** din cadrul funcţiei trebuie să aibă o valoare asociată ei. (Chiar şi în C, dacă o funcţie este declarată ca returnând o valoare, este chiar bine să returneze una.)

Forma generală a instrucţiunii **return** este:

return *expresie*;

Această *expresie* este prezentă doar dacă funcţia este declarată ca returnând o valoare. În acest caz, valoarea *expresiei* va deveni valoarea pe care o va returna funcţia.

Într-o funcţie puteţi să utilizaţi câte instrucţiuni **return** doriţi. Funcţia îşi va încheia execuţia imediat ce va întâlni primul **return**. Acolada care încheie o funcţie determină, de asemenea, ieşirea din funcţie şi este echivalentă cu instrucţiunea **return** neînsoţită de nici o valoare. Dacă apare într-o funcţie care nu este **void**, atunci valoarea returnată este nedefinită.

O funcţie declarată ca **void** poate să nu conţină o instrucţiune **return** care să specifice o valoare. (Atâta vreme cât o funcţie **void** nu poate returna o valoare, este firesc ca nici o instrucţiune **return** dintr-o funcţie **void** să nu poată returna vreo valoare.)

Pentru mai multe informaţii despre **return** vedeţi **Capitolul 6**.

Instrucţiunea *goto*

Deoarece C are un pachet mare de structuri de control şi permite control suplimentar prin **break** şi **continue**, **goto** este puţin necesar. Principala rezervă a celor mai mulţi programatori în ceea ce priveşte instrucţiunea **goto** este tendinţa sa de a crea programe ilizibile. Cu toate acestea, deşi instrucţiunea **goto** a căzut în dizgraţie acum câţiva ani, ea a reuşit într-un fel să îşi mai îmbunătăţească imaginea şifonată. Nu există situaţii în programare care să necesite **goto**; este doar o facilitate care, utilizată înţelept, se face utilă într-un domeniu restrâns de situaţii de programare. Astfel, **goto** nu este utilizată în afara acestui domeniu.

Instrucţiunea **goto** cere o etichetă pentru operaţie. (O *etichetă* este un specificator valid urmat de punct şi virgulă.) Mai mult, eticheta trebuie să se

găsească în aceeaşi funcţie ca şi **goto** care o utilizează, nu puteţi sări între funcţii. Forma generală a instrucţiunii **goto** este:

goto *etichetă*;
.
.
.
etichetă:

unde *etichetă* este orice etichetă validă înainte sau după **goto**. De exemplu, puteţi să creaţi o buclă de la 1 la 100 utilizând **goto** şi o etichetă, aşa cum se prezintă aici:

```
x = 1;
bucla1:
    x++;
    if(x<100) goto bucla1;
```

Instrucţiunea *break*

Instrucţiunea **break** are două utilizări. Puteţi să o utilizaţi pentru a încheia un **case** dintr-o instrucţiune **switch** (după cum s-a discutat mai devreme în acest capitol în secţiunea despre **switch**) sau puteţi să o folosiţi pentru a determina încheierea imediată a unei bucle, trecând peste testul de condiţionare normal.

Când se întâlneşte instrucţiunea **break** într-o buclă, aceasta se încheie imediat iar controlul programului se reia de la instrucţiunea care urmează imediat după buclă. De exemplu:

```
#include <stdio.h>

void main(void)
{
    int t;

    for(t=o; t<100; t++) {
          printf("%d ", t);
          if(t==10) break;
    }
}
```

afişează pe ecran numerele de la 0 la 100. Apoi bucla se termină deoarece **break** determină încheierea imediată a buclei, fără a mai testa condiţia **t<100**.

Programatorii o folosesc deseori în bucle în care este nevoie ca o condiţie specială să ducă la ieşire imediată. De exemplu, aici o apăsare a unei taste poate opri execuţia funcţiei **cauta()**:

```
cauta(char *nume)
{
    do {
            /* cauta nume... */
            if(kbhit ()) break;
    } while(!gasit);
    /* urmarire proces */
}
```

Funcţia **kbhit()** returnează 0 dacă nu apăsaţi o tastă. Altfel, ea returnează o valoare nonzero. Din cauza diferenţelor dintre mediile de calculatoare, nici standardul ANSI C, nici cel propus pentru ANSI C++ nu definesc **kbhit()**, dar aproape sigur o aveţi (pe ea sau una cu nume asemănător) asigurată de compilatorul dvs.

Un **break** determină ieşirea doar din bucla cea mai interioară. De exemplu,

```
for(t=o; t<100: ++t) {
    contor = 1;
    for( ; ; ) {
          printf("%d ", contor);
          contor++;
          if(contor==10) break;
    }
}
```

afişează pe ecran numerele de la 1 la 10 de 100 de ori. De fiecare dată când compilatorul întâlneşte **break**, controlul este trecut înapoi buclei **for** exterioare.

Un **break** utilizat într-o instrucţiune **switch** va afecta doar acel **switch**. El nu afectează nici o buclă în interiorul căreia s-ar afla **switch**.

Funcţia *exit()*

Chiar dacă **exit()** nu este o instrucţiune de control al programului, o scurtă digresiune este acum bine venită. Aşa cum puteţi ieşi dintr-o buclă, la fel puteţi ieşi dintr-un program, utilizând funcţia **exit()** din biblioteca standard. Această funcţie determină încheierea imediată a unui întreg program, forţând reîntoarcerea în sistemul de operare. Acţionează, de fapt, ca un **break** generalizat.

Forma generală a funcţiei **exit()** este:

void exit (int *cod_de_intoarcere*);

Valoarea lui *cod_de_intoarcere* este returnată procesului care a apelat, care este, de obicei, sistemul de operare. Zero este utilizat în general pentru a returna un cod care indică terminarea normală a programului. Alte argumente sunt utilizate pentru a indica unele tipuri de erori.

Programatorii utilizează deseori **exit()** atunci când nu este îndeplinită o anumită condiție obligatorie pentru program. De exemplu, imaginați-vă un joc pe calculator în realitate virtuală care necesită un adaptor grafic special. Funcția **main()** a acestui joc poate să arate astfel:

```
void main(void)
{
    if(!grafica_virtuala()) exit(1);
    play();
}
```

unde **grafica_virtuala()** este o funcție definită de utilizator care returnează adevărat dacă există adaptorul de grafică virtuală. Dacă adaptorul nu este în sistem, **grafica_virtuala** returnează fals iar programul se încheie.

Un alt exemplu este această versiune pentru **menu()** care folosește **exit()** pentru ieșirea din program și reîntoarcerea în sistemul de operare:

```
void menu(void)
{
    char ch;

    printf("1. Control ortografic\n");
    printf("2. Corecteaza erorile de ortografie\n");
    printf("3. Afiseaza erorile de ortografie\n");
    printf("Iesire\n");
    printf("     Introduceti o optiune: ");

    do {
         ch = getchar(); /* citeste selectia de la
                             tastatura */
            switch(ch) {
                case '1':
                    control_orto();
                    break;
                case '2':
                    corect_erori();
                    break;
                case '3':
                    afis_erori();
                    break;
                case '4':
                    exit(0);  /* reintoarcere in OS */
            }
    } while(ch!='1' && ch!= '2' && ch!='3');
}
```


Instrucțiunea *continue*

Instrucțiunea **continue** lucrează într-un fel ca și instrucțiunea **break**. În loc însă să forțeze încheierea, **continue** forțează trecerea la următoarea iterație a buclei, ignorând restul codului iterației în care se află. În bucla **for**, **continue** determină execuția testului de condiționare și apoi a secvenței de incrementare. Pentru buclele **while** și **do-while** controlul programului este trecut testului de condiționare. De exemplu, următorul program numără câte spații sunt conținute în șirul introdus de către utilizator:

```
/* Numara spatiile */
#include <stdio.h>

void main(void)
{
    char s[80], *sir;
    int spatiu;

    printf("introduceti un sir: ");
    gets(s);
    sir = s;

    for(spatiu=0; *sir; sir++) {
         if(*sir != ' ') continue;
         spatiu++;
    }
    printf("%d spatii\n", spatiu);
}
```

Fiecare caracter este testat pentru a se vedea dacă este un spațiu. Dacă nu este, instrucțiunea **continue** forțează bucla **for** să treacă la o nouă iterație. Dacă el *este* un spațiu, este incrementată variabila **spatiu**.

Următorul exemplu prezintă modul de utilizare a lui **continue** pentru a grăbi ieșirea dintr-o buclă forțând testul de condiționare să fie efectuat mai repede:

```
void cod(void)
{
    char gata, ch;

    gata = 0;
    while(!gata) {
          ch=getchar();
          if(ch=='$') {
                gata = 1;
                continue;
          }
          putchar(ch+1); /* preia litera urmatoare din
                                  alfabet */
    }
}
```

Această funcție codifică un mesaj schimbând toate caracterele pe care le tastați cu litera următoare. De exemplu, A devine B. Funcția se va încheia când scrieți **$**. După ce a fost introdus **$**, nu va mai apărea nici o ieșire deoarece testul de condiționare activat de **continue** va găsi **gata** adevărat, ceea ce va determina părăsirea buclei.

Instrucțiuni de tip expresie

Capitolul 2 acoperă în întregime expresiile. Totuși, aici mai trebuie menționate câteva lucruri. Amintiți-vă că o instrucțiune de tip expresie este pur și simplu o expresie validă urmată de punct și virgulă, ca în:

```
func();  /* o apelare de functie */
a = b+c; /* o instructiune de atribuire */
b+f();   /* o instructiune valida dar stranie */
;        /* o instrucţiune vida */
```

Prima instrucțiune de tip expresie este o apelare de funcție. A doua este una de atribuire. A treia expresie, cea stranie, este totuși evaluată de compilatorul C/C++ deoarece funcția **f()** poate să efectueze o anumită sarcină. Ultimul exemplu arată că C permite ca o instrucțiune să fie vidă (uneori numită *instrucțiune nulă*).

Instrucțiuni bloc

Instrucțiunile *bloc* sunt simple grupuri de instrucțiuni înrudite care sunt tratate ca o unitate. Instrucțiunile care formează un bloc sunt unite logic. Un bloc începe cu o acoladă deschisă ({) și se încheie cu corespondentul său, acolada închisă (}).

Cel mai adesea programatorii folosesc instrucțiunile bloc pentru a crea o instrucțiune multiplă ca obiect al unei alte instrucțiuni, cum ar fi **if**. Totuși, puteți să plasați o instrucțiune bloc oriunde ați putea introduce orice altă instrucțiune. De exemplu, acesta este un cod în C perfect valid (deși neobișnuit):

```
#include <stdio.h>

void main(void)
{
    int i;

    {      /* o instructiune bloc */
           i = 120;
           printf("%d", i);
    }
}
```

Capitolul 4

Matrice şi şiruri



O *matrice* este o colecție de variabile de același tip, apelate cu același nume. Accesul la un anumit element al matricei se face cu ajutorul unui indice. În C, toate matricele constau în locații de memorie contigue. Cel mai mic indice corespunde primului element iar cel mai mare ultimului element. Matricele pot avea una sau mai multe dimensiuni. Cea mai simplă matrice în C este *șirul*, care este o matrice de caractere terminate cu un caracter null. Această caracteristică a șirurilor oferă limbajului C mai multă putere și eficiență decât posedă alte limbaje.

În C, matricele și pointerii sunt strâns legați; o discuție despre unele se referă de obicei și la ceilalți. Acest capitol se axează pe matrice, în timp ce **Capitolul 5** se ocupă îndeaproape de pointeri. Trebuie să le citiți pe amândouă pentru a înțelege pe deplin aceste construcții importante în C.

Matrice cu o singură dimensiune

Forma generală pentru declararea unei matrice cu o singură dimensiune este:

tip nume_variab[mărime];

Ca și alte variabile, matricele trebuie declarate explicit astfel încât compilatorul să aloce spațiu în memorie pentru ele. Aici, *tip* declară tipul de bază al matricei, care este tipul fiecărui element al său. *mărime* indică ce număr de elemente va conține matricea. De exemplu, pentru a declara o matrice cu 100 de elemente numită **bilant**, de tipul **double**, se utilizează această instrucțiune:

```
double bilant[100];
```

În C, toate matricele au 0 ca indice pentru primul element. De aceea, când scrieți

```
char p[10];
```

declarați o matrice de caractere care are zece elemente, de la **p[0]** la **p[9]**. De exemplu, următorul program încarcă o matrice de întregi cu numerele de la 0 la 99.

```
void main(void)
{
    int x[100]; /* declara o matrice cu 100 intregi */
    int t;

    for(t=0; t<100;  ++t) x[t] = t;
}
```


Cantitatea de memorie necesară pentru înregistrarea unei matrice este direct proporțională cu tipul și mărimea sa. Pentru o matrice unidimensională, mărimea totală în octeți este calculată astfel:

*total octeți = sizeof(tip de baza) * marimea matricei*

C nu controlează limitele unei matrice. Puteți depăși ambele margini ale unei matrice și scrie în locul altor variabile sau peste codul programului. Este treaba dvs. ca programator să asigurați controlul limitelor acolo unde este necesar. De exemplu, acest cod va fi compilat fără eroare, dar este incorect deoarece bucla **for** va determina ca matricea **numara** să-și depășească limitele:

```
int count[100], i;
/* aceasta determina numara sa depaseasca limitele */
for(i=0; i<100; i++) numara[i] = i;
```

Matricele unidimensionale sunt de fapt liste de informații de același tip care sunt stocate în locații de memorie contigue în ordinea indicilor. De exemplu, **Figura 4-1** arată cum apare în memorie matricea a dacă începe la locația de memorie 1000 și este declarată astfel:

```
char a[7];
```

Crearea unui pointer la o matrice

Un pointer la primul element al matricei se creează simplu, specificând numele matricei, fără nici un indice. De exemplu, având

```
int proba[10];
```

puteți să creați un pointer la primul ei element utilizând numele **proba**. Următorul fragment de program atribuie lui **p** adresa primului element din **proba**.

```
int *p;
int proba[10];

p = proba;
```

Element	a[0]	a[1]	a[2]	a[3]	a[4]	a[5]	a[6]
Adresa	1000	1001	1002	1003	1004	1005	1006

Figura 4-1 *O matrice de șapte caractere care începe de la locația 1000*

Puteţi, de asemenea, să specificaţi adresa primului element al unei matrice utilizând operatorul &. De exemplu, **proba** şi **&proba[0]** au acelaşi rezultat. Totuşi, într-un cod C/C++ profesionist, nu veţi vedea aproape niciodată **&proba[0]**.

Transmiterea matricelor unidimensionale către funcţii

În C nu puteţi transmite o matrice întreagă ca argument al unei funcţii. Puteţi, totuşi, să introduceţi un pointer la o matrice specificând numele acesteia fără indice. De exemplu, următorul fragment de program introduce adresa lui **i** în funcţia **func1()**.

```
void main(void)
{
    int i[10];

    func1(i);
    .
    .
    .
}
```

Dacă funcţia primeşte o matrice unidimensională, puteţi să declaraţi parametrul său formal în unul dintre aceste trei moduri: ca pointer, ca matrice cu dimensiune sau ca matrice fără dimensiune. De exemplu, pentru a primi pe **i** funcţia cu numele **func1()** poate fi declarată astfel:

```
void fun(int *x) /* pointer */
{
    .
    .
    .
}
```

sau

```
void func1(int x[10]) /* matrice cu dimensiune */
{
    .
    .
    .
}
```


sau, în sfârşit,

```
void func1(int x[]) /* matrice fara dimensiune */
{
    .
    .
    .
}
```

Toate trei metodele de declarare determină rezultate similare deoarece fiecare spune compilatorului că va fi primit un pointer către un întreg. Prima declaraţie foloseşte chiar un pointer. Cea de a doua lucrează cu declaraţia de matrice standard. În ultima variantă, o versiune modificată a unei declaraţii de matrice, se specifică pur şi simplu că va fi introdusă o matrice de tip **int** şi de o anumită mărime. După cum puteţi vedea, mărimea matricei nu contează pentru funcţie, deoarece C nu efectuează controlul limitelor. De fapt, din punctul de vedere al compilatorului, va fi corectă şi forma:

```
void func1(int x[32])
{
    .
    .
    .
}
```

deoarece compilatorul de C creează un cod care instruieşte **func1()** să primească un pointer - el nu creează efectiv o matrice cu 32 de elemente.

Şiruri

De departe cea mai utilizată matrice unidimensională este şirul de caractere. Amintiţi-vă că în C un şir este definit ca o matrice de caractere care se termină cu un caracter null. Un null este specificat ca '\0' şi are valoarea 0. Din acest motiv, trebuie să declaraţi matricele de tip caracter cu un caracter mai mult decât cel mai mare şir pe care îl vor conţine. De exemplu, pentru a declara matricea **sir** care poate conţine 10 caractere, veţi scrie scrie:

```
char sir[11];
```

Astfel, se face loc şi pentru caracterul null de la sfârşitul şirului.

Deşi C nu are date de tip şir, el permite constante şir. O *constantă şir* este o listă de caractere închise între ghilimele simple. De exemplu,

```
'salut'
```

Nu este necesar să introduceți manual caracterul null la sfârșitul șirului de constante - compilatorul o face automat, pentru dvs..

C admite o gamă largă de funcții de manevrare a șirurilor. Cele mai uzuale sunt prezentate aici:

Nume	Funcție
strcpy(s1, s2)	Copiază s1 în s2
strcat(s1, s2)	Concatenează s2 la sfârșitul lui s1
strlen(s1)	Returnează lungimea lui s1
strcmp(s1, s2)	Returnează 0 dacă s1 și s2 sunt identice; mai mic decât 0 dacă s1<s2; mai mare ca 0 dacă s1>s2
strchr(s1, ch)	Returnează un pointer la prima apariție a lui ch în s1
strstr(s1, s2)	Returnează un pointer la prima apariție a lui s2 în s1

Aceste funcții utilizează fișierul antet standard STRING.H. Următorul program demonstrează cum pot fi utilizate pentru șiruri.

```
#include <stdio.h>
#include <string.h>

void main(void)
{
    char s1[80], s2[80];

    gets(s1);
    gets(s2);

    printf("lungimi: %d %d\n", strlen(s1), strlen(s2));
    if(!strcmp(s1, s2)) printf("Sirurile sunt egale\n");
    strcat(s1, s2);
    printf("%s\n", s1);
    strcpy(s1, "Acesta este un test.\n");
    printf(s1);
    if(strchr("hello", 'e')) printf("e este in hello\n");
    if(strstr("te salut", "te")) printf("am gasit te");
}
```

Dacă rulați acest program și introduceți șirurile "**hello**" și "**hello**", ieșirea este:

lungimi: 5 5
Sirurile sunt egale
hellohello
Acesta este un test.
e este in hello
am gasit te



*REȚINEȚI: **strcmp()** returnează fals dacă șirurile sunt egale. Dacă verificați egalitatea, fiți siguri că utilizați operatorul logic ! pentru a obține inversul condiției, așa cum s-a arătat.*

Matrice bidimensionale

C admite matrice multidimensionale. Cea mai simplă formă de matrice multidimensională este cea bidimensională. O matrice bidimensională este de fapt o matrice de matrice unidimensionale. Pentru a declara o matrice bidimensională de întregi numită **d**, de mărime 10,20, veți scrie:

```
int d[10][20];
```

Fiți foarte atenți la declarare. Majoritatea celorlalte limbaje folosesc virgule pentru a separa dimensiunile matricei; C plasează fiecare dimensiune în câte o pereche de paranteze drepte.

Similar, pentru a avea acces la punctul 1,2 al matricei **d**, veți utiliza:

```
d[1][2]
```

Următorul exemplu încarcă numere de la 1 la 12 într-o matrice bidimensională și le afișează rând cu rând.

```
#include <stdio.h>

void main(void)
{
    int t, i, num[3][4];

    for(t=0; t<3; ++t)
        for(i=0; 1<4; ++i)
            num[t][i] = (t*4)+i+1;
    /* acum le afiseaza */
    for(t=0; t<3; ++t) {
        for(i=0; i<4; ++i)
            printf("%3d ", num[t][i]);
        printf("\n");
    }
}
```

În acest exemplu, **num[0][0]** are valoarea 1, **num[0][1]** valoarea 2, **num[0][2]** valoarea 3 şi aşa mai departe. Valoarea lui **num[2][3]** va fi 12. Puteţi să vizualizaţi matricea **num** în prezentarea de mai jos:

num [t] [i]

	0	1	2	3
0	1	2	3	4
1	5	6	7	8
2	9	10	11	12

Matricele bidimensionale sunt stocate în forma rând-coloană, unde primul indice indică rândul iar al doilea precizează coloana. Aceasta înseamnă că indicele din dreapta se modifică mai repede decât cel din stânga atunci când parcurgeţi elementele matricei în ordinea în care sunt stocate efectiv în memorie. Pentru o reprezentare grafică a unei matrice bidimensionale priviţi **Figura 4-2**.

În cazul unei matrice bidimensionale următoarea formulă calculează numărul de octeţi din memorie necesari pentru a o reţine:

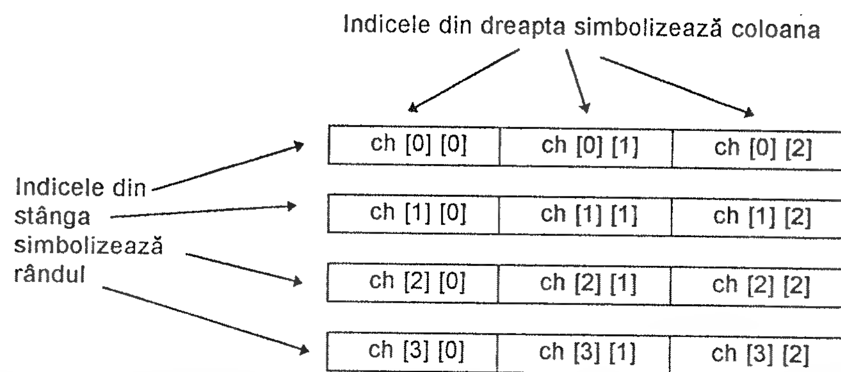


Figura 4-2 *O matrice bidimensională în memorie*

*octeţi = mărimea primului indice * mărimea celui de-al doilea indice * sizeof (tip de dată)*

De aceea, presupunând că un întreg este înregistrat pe doi octeţi, o matrice de întregi cu dimensiunile 10,5 va avea alocaţi

10 * 5 * 2

adică 100 octeţi.

Când o matrice bidimensională este utilizată ca un argument pentru o funcţie, este transmis doar un pointer către primul element al matricei. Însă, parametrul care care primeşte o matrice bidimensională trebuie să definească cel puţin numărul de coloane din dreapta deoarece compilatorul de C/C++ trebuie să ştie lungimea fiecărui rând pentru a indexa corect matricea. De exemplu, o funcţie care primeşte o matrice bidimensională de întregi cu dimensiunea 10, 10 este declarată astfel:

```
void func1(int x[][10])
{
      .
      .
      .
}
```

Puteţi specifica şi dimensiunea din stânga dacă doriţi, dar nu este necesar. În ambele cazuri, compilatorul trebuie să ştie mărimea celei din dreapta pentru a executa corect expresii ca:

```
x[2][4]
```

în interiorul funcţiei. Dacă mărimea rândului nu se cunoaşte, compilatorul nu poate să determine unde începe al treilea rând.

Următorul scurt program utilizează o matrice bidimensională pentru a memora notele fiecărui student din clasele unui profesor. Programul presupune că profesorul are trei clase şi un maxim de 30 de studenţi în fiecare clasă. Reţineţi modul în care matricea **note** este parcursă de fiecare dintre funcţii.

```
#include <stdio.h>
#include <ctype.h>
#include <stdlib.h>

/* O baza de date simpla pentru notele studentilor */
```

```
#define CLASE 3
#define NOTE 30

int note[CLASE][NOTE];

void introd_note(void);
int preia_note(int num);
void afis_note (int g[][NOTE]);

void main(void)
{
    char ch, sir[80];

    for(;;) {
          do {
                printf("(I)ntroduceti notele\n");
                printf("(P)rezinta notele\n");
                printf("(Q)uit\n");
                gets(sir);
                ch = mari(*sir);
          } while(ch!='E' && ch!='R' && ch!='Q');

          switch(ch) {
                case 'E':
                      introd_note();
                      break;
                case 'R':
                      afis_note(note);
                      break;
                case 'Q':
                      iesire(0);
          }
    }
}

/* Introduceti notele studentilor. */
void introd_note(void)
{
    int t, i;

    for(t=0; t<CLASE; t++) {
          printf("Clasa # %d:\n", t+1);
          for(i=0; i<NOTE; ++i)
                      note[t][i] = preia_note(i);
          }
    }

/* Citeste o nota. */
preia_note(int num)
{
    char s[80];

    printf("Introduceti nota pentru studentul # %d:\n",
            num+1);
    gets(s);
    return(atoi(s));
}

/* Afiseaza notele. */
void afis_note(int g[][NOTE])
{
    int t, i;

    for(t=0; t<CLASE; ++t)   {
          printf("Clasa # %d:\n", t+1);
          for(i=0; i<NOTE; ++i)
                printf("Student #%d este %d\n", i+1,
                        g[t][i]);
    }
}
```


Matrice de şiruri

Utilizarea în programare a matricelor de şiruri nu este neobişnuită. De exemplu, procesorul de intrare într-o bază de date poate să compare comenzile date de utilizator cu o matrice de comenzi. Pentru a crea o matrice de şiruri, utilizaţi o matrice bidimensională de tip caracter. Mărimea indicelui din stânga determină numărul de şiruri iar mărimea indicelui din dreapta specifică mărimea maximă a fiecărui şir. Următorul cod declară o matrice cu 30 de şiruri, fiecare din ele cu lungimea maximă de 79 de caractere.

```
char matrice_siruri[30][80];
```

Este uşor de căpătat acces la un şir individual: specificaţi pur şi simplu doar indicele din stânga. De exemplu, următoarea instrucţiune apelează **gets()** pentru al treilea şir din **matrice_siruri**.

```
gets(matrice-siruri[2]);
```

Precedenta instrucţiune este echivalentă funcţional cu:

```
gets(&matrice_siruri[2][0]);
```

dar prima dintre forme este mai uzuală pentru codul C/C++ profesionist.

Pentru a înţelege mai bine cum lucrează matricele de şiruri, studiaţi următorul scurt program care foloseşte o matrice de şiruri ca bază pentru un editor de texte foarte simplu.

```
#include <stdio.h>

#define MAX 100
#define LUNG 80

char text[MAX][LUNG];

/* Un editor de text foarte simplu. */
void main(void)
{
    register int t, i, j;

    printf("Introduceti o linie goala pentru a iesi.\n");

    for(t=0; t<MAX; t++)  {
          printf("%d: ", t);
          gets(text[t]);
          if(!*text[t]) break; /* iesire din program la
                                  prima linie libera */
    }

    for(i=0; i<t; i++) {
          for(j=0; text[i][j]; j++) putchar(text[i][j]);
          putchar('\n');
    }
}
```

Acest program introduce linii de text până când se dă o linie goală. Apoi el reafişează câte un caracter o dată pe fiecare linie.

Matrice multidimensionale

C permite matrice cu mai mult de două dimensiuni. Limita exactă, dacă există una, este determinată de compilatorul dvs. Forma generală de declarare a unei matrice multidimensionale este:

tip nume[Mărime1][Mărime2][Mărime3]...[MărimeN];

Matricele cu mai mult de două dimensiuni nu sunt utilizate prea des datorită cantităţii de memorie cerute. De exemplu, o matrice de caractere cu patru dimensiuni de mărime 10, 6, 9, 4 necesită

10 * 6 * 9 * 4

deci 2.160 octeţi. Dacă matricea reţine întregii în doi octeţi vor fi necesari 4.320 octeţi. Dacă matricea memorează date de tip **double** (presupunând 8 octeţi pentru fiecare **double**), vor fi necesari 17.280 de octeţi. Memoria necesară creşte exponenţial cu numărul dimensiunilor.

În matricele multidimensionale, calculatorului îi ia un timp pentru a prelucra indicii. Aceasta înseamnă că accesul la un element dintr-o matrice multidimensională poate fi mai lent decât accesul la un element dintr-o matrice cu o singură dimensiune.

Când transmiteţi o matrice multidimensională unei funcţii, trebuie să declaraţi toate dimensiunile, în afară de cea din extrema stângă. De exemplu, dacă declaraţi o matrice cu **m** dimensiuni ca aceasta:

```
int m[4][3][6][5];
```

o funcţie **func1()** care primeşte pe **m** va arăta astfel:

```
void func1(int d[][3][6][5])
{
     .
     .
     .
}
```

Desigur, dacă doriţi, puteţi să introduceţi prima dimensiune.

Pointeri de indexare

În C, pointerii şi matricele sunt strâns legaţi. După cum ştiţi, numele unei matrice fără un indice este un pointer la primul element al matricei. De exemplu, să luăm

această matrice:

```
char p[10];
```

Următoarele instrucțiuni sunt identice cu:

```
p
&p[0]
```

Altfel spus,

```
p == &p[0]
```

este evaluat ca adevărat deoarece adresa primului element al unei matrice este aceeași cu adresa matricei.

După cum am mai spus, numele unei matrice fără un indice generează un pointer. Invers, un pointer poate să aibă un indice ca și cum ar fi fost declarat ca o matrice. De exemplu, să luăm acest mic fragment:

```
int *p, i[10];
p = i;
p[5] = 100;  /* atribuire utilizind indice */
*(p+5) = 100; /* atribuire utilizind aritmetica pointerilor */
```

Ambele instrucțiuni de atribuire plasează valoarea 100 în cel de-al șaselea element al lui **i**. Prima instrucțiune filoseşte un indice al lui **p**; cea de-a doua utilizează aritmetica pointerilor. În oricare mod rezultatul este acelaşi. (**Capitolul 5** discută pointerii și aritmetica lor.)

Același concept se aplică la matricele de două sau mai multe dimensiuni. De exemplu, presupunând că **a** este o matrice de întregi de 10 pe 10, aceste două instrucțiuni sunt echivalente:

```
a
&a[0][0]
```

Mai mult, la elementul 0, 4 al lui **a** se poate face referire în două moduri: ori prin indecșii matricei **a[0][4]**, ori prin pointerul ***(a+4)**. Similar, elementul 1, 2 este ori **a[1][2]** ori ***(a+12)**. În general, pentru orice matrice bidimensională

a[i][k]

este echivalent cu

*(a+(j**lungime rând*)+k)_

Pointerii sunt uneori folosiți pentru a avea acces în matrice deoarece aritmetica pointerilor este în general mai rapidă decât adăugarea de indici matricei.

O matrice bidimensională poate fi redusă la un pointer la o matrice unidimensională. De aceea, utilizarea unei variabile de tip pointer separată este o cale ușoară de a folosi pointerii pentru a avea acces la elementele dintr-un rând al unei matrice bidimensionale. Următoarea funcție ilustrează această tehnică. Ea va afișa conținutul rândului specificat al matricei de variabile globale de tip întreg **num**.

```
int num[10][10];
.
.
.
void afis_rind(int j)
{
    int *p, t;

    p = &num[j][0]; /* preia adresa primului element al
                       rindului j */
    for(t=0; t<10; ++t) printf("%d ", *(p+t));
}
```

Puteți generaliza această rutină stabilind ca argumente de apelare rândul, lungimea sa și un pointer la primul element al matricei, așa cum se arată aici:

```
void afis_rind(int j, int dim_rind, int *p)
{
    int t;

    p = p + (j * dim_rind);

    for(t=0; t<dim_rind; ++t)
        printf("%d ", *(p+t));
}
```

Matricele mai mari de două dimensiuni pot fi reduse într-un mod similar. De exemplu, o matrice cu trei dimensiuni poate fi redusă la un pointer la o matrice bidimensională care poate fi redusă la un pointer la o matrice unidimensională. În general, o matrice *n*-dimensională poate fi redusă la un pointer la o matrice cu (*n*-1) dimensiuni. Această nouă matrice poate fi redusă prin aceeași metodă. Procesul se încheie când se ajunge la o matrice unidimensională.

Iniţializarea matricelor

C permite iniţializarea matricelor în acelaşi timp cu declararea lor. Forma generală de iniţializare este similară cu aceea a celorlalte variabile, aşa cum se prezintă aici:

Specificator_de_tip nume_matrice[*marime1*]....[*marimeN*] = {*listă_de_valori*};

Lista _de_valori este o listă separată prin virgule de un tip compatibil cu *specificator_de_tip*. Prima constantă este plasată în prima poziţie a matricei, a doua constantă în a doua poziţie şi aşa mai departe. Reţineţi că } este urmată de punct şi virgulă.

În următorul exemplu o matrice de întregi cu 10 elemente este iniţializată cu numerele de la 1 la 10:

```
int i[10] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
```

Aceasta înseamnă că **i[0]** va avea valoarea 1 iar **i[9]** va avea valoarea 10.

Matricele de caractere care conţin şiruri permit o iniţializare prescurtată care are forma:

char *nume_matrice*[*mărime*] = "*şir*";

De exemplu, următorul fragment de cod iniţializează **sir** cu fraza „Imi place C++".

```
char str[14] = "Imi place C++";
```

Aceasta este similar cu a scrie:

```
char sir[14] = {'I', 'm', 'i', ' ', 'p', 'l', 'a', 'c',
                'e', ' ', 'C', '+', '+', '\0'};
```

Deoarece toate şirurile în C se termină cu null, trebuie să fiţi sigur că matricea pe care o declaraţi este suficient de lungă pentru a-l include. Din acest motiv, **sir** are mărimea de 14 caractere chiar dacă „Imi place C++" are doar 13. Când folosiţi constante de tip şir, compilatorul asigură automat caracterul de null de sfârşit.

Matricele multidimensionale sunt iniţializate la fel cu cele cu o singură dimensiune. De exemplu, următorul cod iniţializează **patrat** cu numerele de la 1 la 10 şi cu pătratele lor.


```
int patrat[10][2] = {
    1,1,
    2,4,
    3,9,
    4,16,
    5,25,
    6,36,
    7,49,
    8,64,
    9,81,
    10,100
};
```

Iniţializarea matricelor fără mărime

Imaginaţi-vă că folosiţi o iniţializare pentru a construi un tabel de mesaje de eroare, astfel:

```
char e1[18] = "Eroare de citire\n";
char e2[19] = "Eroare de scriere\n";
char e3[28] = "Nu pot sa deschid fisierul\n";
```

După cum vă puteţi da seama, este greu de numărat manual caracterele din fiecare mesaj pentru a determina mărimea exactă a matricei. Puteţi lăsa compilatorul să calculeze singur această mărime utilizând matricele cu mărime nedeterminată. Dacă într-o instrucţiune de iniţializare a unei matrice nu este specificată mărimea sa, compilatorul C/C++ creează automat o matrice suficient de mare pentru a păstra toate iniţializările existente. Ea se numeşte o *matrice fără dimensiune*. Dacă folosiţi această facilitate tabelul de mesaje devine:

```
char e1[] = "Eroare de citire\n";
char e2[] = "Eroare de scriere\n";
char e3[] = "Nu pot sa deschid fisierul\n";
```

Cu aceste iniţializări următoarea instrucţiune

```
printf("%s are marimea %d\n", e2, sizeof e2);
```

va afişa

Eroare de citire are marimea 18

În afară de faptul că este mai simplă, iniţializarea matricei fără dimensiune vă permite schimbarea oricărui mesaj fără teamă că utilizaţi incorect dimensiunile

matricei.

Iniţializarea matricelor fără dimensiune nu este restrânsă la matricele unidimensionale. Pentru cele multidimensionale trebuie să specificaţi toate dimensiunile în afară de cea din extrema stângă. (Celelalte dimensiuni sunt necesare pentru a permite compilatorului C/C++ să aloce corect indici matricei.) În acest fel puteţi crea tabele de mărimi diferite iar compilatorul va aloca automat suficientă memorie pentru ele. De exemplu, aici este prezentată declararea lui **patrat** ca o matrice fără dimensiune:

```
int patrat[][2] = {
    1,1,
    2,4,
    3,9,
    4,16,
    5,25,
    6,36,
    7,49,
    8,64,
    9,81,
    10,100
};
```

Alt avantaj al acestei declaraţii, în afară de cel în legătură cu mărimea, este că puteţi mări sau scurta tabelul fără să modificaţi dimensiunile matricei.

Un exemplu de joc „O şi X"

Exemplul lung care urmează ilustrează multe dintre căile în care puteţi manevra matricele în C. Matricele bidimensionale sunt folosite curent pentru a simula jocurile de pe tabele matriciale. Această secţiune prezintă un simplu program pentru „O şi X".

Calculatorul joacă un joc foarte simplu. Când este rândul său la mutare el utilizează **muta_calculatorul()** pentru a parcurge matricea în căutarea unei căsuţe libere. Când găseşte una, pune un **O** în ea. Dacă nu poate găsi un loc liber, declară jocul pierdut şi iese din program. Funcţia **muta_jucatorul()** vă întreabă unde doriţi să puneţi un **X**. Colţul din stânga sus are poziţia 1,1; cel din dreapta jos, 3,3.

Matricea este iniţializată cu spaţii. Fiecare mişcare făcută de jucător sau de calculator schimbă spaţiul cu un X sau cu un O, ceea ce permite cu uşurinţă afişarea matricei pe ecran.

De fiecare dată când se efectuează câte o mutare programul apelează funcţia **verifica()**. Ea returnează un spaţiu dacă nu este încă nici un învingător, un X dacă aţi câştigat dvs. sau un O dacă a câştigat calculatorul. Parcurge rândurile, coloanele şi apoi diagonalele, căutând dacă vreuna conţine doar X-uri sau doar O-uri.

Funcţia **afis_matrice()** afişează stadiul curent al jocului. Observaţi cum modul de iniţializare a matricei simplifică această funcţie.

Rutinele din acest exemplu au acces la matricea **matrice** în mod diferit. Studiaţi-le pentru a vă asigura că aţi înţeles fiecare operaţie cu matrice.

```
/* Un simplu joc de 0 si X. */
#include <stdio.h>
#include <stdlib.h>

char matrice[3][3]; /* matricea 0 si X */

char verifica(void);
void matrice_init(void)
void muta_jucatorul(void);
void muta_calculatorul(void);
void afis_matrice(void);

void main(void)
{
    char gata;

    printf("Acesta este jocul ) si X.\n");
    printf("Veti juca cu calculatorul.\n");

    gata = ' ';
    matrice_init();
    do{
          afis_matrice();
          muta_jucatorul();
          gata = verifica(); /* verifica daca exista
                                invingator */
          if(gata!= ' ') break; /* castigator */
          muta_calculator();
          gata = verifica(); /* verifica daca exista
                                invingator */
    }     while(gata== ' ');
    if(gata== 'X') printf("Ati cistigat!\n");
    else printf("Am cistigat!!!\n");
    afis_matrice(); /* arata pozitia finala */
}
/* Initializeaza matricea */
```

```
void matrice_init(void)
{
    int i, j;

    for(i=0; i<3; i++)
          for(j=0; j<3; j++) matrice[i][j] = ' ';
}

/* Muta jucatorul. */
void muta_jucatorul(void)
{
    int x, y;

    printf("Introduceti coordonatele pentru X-ul dvs.: ");
    scanf("%d%d", &x, &y);

    x--; y--;

    if(matrice[x][y]!= ' '){
          printf("Mutare incorecta, mai incercati. \n");
          muta_jucatorul();
    }
    else matrice[x][y] = 'X'

}
/* Muta calculatorul */
void muta_calculatorul(void)
{
    int i, j;

    for(i=0; i<3; i++){
          for(j=0; j<3; j++)
                if(matrice[i][j]==' ') break;
          if(matrice[i][j]==' ') break;
    }

    if(i*j==9) {
          printf("gata\n");
          exit(0);
    }
    else
          matrice[i][j] = 'O';
}
```



```
/* Afiseaza matricea pe ecran. */
void afis_matrice(void)
{
    int t;

    for(t=0; t<3; t++) {
          printf(" %c | %c | %c ", matrice[t][0],
                matrice[t][1], matrice[t][2]);
          if(t!=2) printf("\n---|---|---\n");
    }
    printf("\n");
}

/* Verifica daca exista invingator. */
char verifica(void)
{
    int i;

    for(i=0; i<3; i++) /* verifica rindurile */
          if(matrice[i][0]==matrice[i][1] &&
             matrice[i][0]==matrice[i][2])
             return matrice[i][0];
    for(i=0; i<3; i++) /* verifica coloanele */
          if(matrice[0][i]==matrice[1][i] &&
             matrice[0][i]==matrice[2][i],
             return matrice[0][i];
/* verifica diagonalele */
if(matrice[0][0]==matrice[1][1] &&
    matrice[1][1]==matrice[2][2])
    return matrice [0][0];

if(matrice[0][2]==matrice[1][1] &&
   matrice[1][1]==matrice[2][0])
   return matrice[0][2];
return ' ';
}
```

Capitolul 5

Pointeri



Înțelegerea și utilizarea corectă a pointerilor este esențială pentru succesul programării în C (și C++). Pentru aceasta există trei motive: primul, pointerii oferă posibilitatea de a modifica argumentele de apelare a funcțiilor. Al doilea, pointerii permit o alocare dinamică. Al treilea, pointerii pot îmbunătăți eficiența anumitor rutine. De asemenea, după cum veți vedea în **Partea a doua**, în C++ pointerii mai au și alte roluri, importante.

Pointerii sunt una dintre cele mai puternice caracteristici ale lui C dar și cele mai periculoase. De exemplu, pointerii neinițializați (sau pointerii care conțin valori neadecvate) pot determina blocarea sistemului. Încă și mai rău, este ușor să fie folosiți incorect, implicând erori foarte greu de depistat.

Atât datorită importanței cât și potențialului de erori, acest capitol examinează în detaliu subiectul pointerilor.

Ce sunt pointerii?

Un *pointer* este o variabilă care conține o adresă din memorie. Această adresă este localizarea în memorie a unui alt obiect (de obicei o altă variabilă). De exemplu, dacă o variabilă conține adresa alteia, prima se spune că este *un pointer la* (indică pe) cea de a doua. **Figura 5-1** ilustrează aceasta.

Variabile de tip pointer

Dacă o variabilă urmează să rețină un pointer, el trebuie declarat ca atare. O declarare de pointeri constă dintr-un tip de bază, un * și numele variabilei. Forma generală de declarare a unui pointer este:

*tip * nume*;

unde *tip* este tipul de bază al pointerului și poate fi orice nume valid. Numele variabilei de tip pointer este specificat prin *nume*.

Tipul de bază al pointerului definește tipul de variabilă către care indică acesta. Practic, orice tip de pointer poate să indice orice în memorie. Totuși, toată aritmetica pointerilor este creată relativ la tipul lor de bază, astfel încât este important să declarați corect pointerul. (Aritmetica pointerilor este prezentată în continuare în acest capitol.)

Operatori pentru pointeri

Există doi operatori speciali pentru pointeri: * și &. **&** este un operator unar care returnează adresa din memorie a operandului său. (Amintiți-vă că un operator unar necesită doar un singur operand.) De exemplu,

```
m = &numara;
```


Adresă de memorie	Variabila din memorie
1000	1003
1001	
1002	
1003	
1004	
1005	
1006	

Memorie

Figura 5-1 *O variabilă indică spre alta*

introduce în **m** adresa variabilei **numara**. Această adresă este locația internă din calculator a variabilei. Ea nu are nici o legătură cu valoarea din **numara**. Puteți să considerați **&** ca returnând „adresa lui". De aceea, instrucțiunea de atribuire precedentă înseamnă „**m** primește adresa lui **numara**".

Pentru a înțelege mai bine atribuirea anterioară, presupuneți că variabila **numara** folosește locația 2000 pentru a memora valoarea sa. Mai presupuneți că **numara** are valoarea 100. După atribuirea precedentă, **m** va avea valoarea 2000.

Al doilea operator pentru pointeri, *****, este complementul lui **&**. El este un operator unar care returnează valoarea înregistrată la adresa care îi urmează. De exemplu, dacă **m** conține adresa din memorie a variabilei **numara**,

```
q = *m;
```

plasează valoarea din **numara** în **q**. De aceea, **q** va avea valoarea 100, deoarece 100 este stocat la locația 2000, care este adresa din memorie care a fost stocată în

m. Vă puteţi gândi la * ca fiind „de la adresa". În acest caz, instrucţiunea precedentă înseamnă „q primeşte valoarea de la adresa m."

Uneori, pentru începători, produce confuzie faptul că acelaşi semn este şi pentru înmulţire şi pentru „la adresa" iar semnul pentru AND asupra biţilor este acelaşi şi pentru „adresa lui". Aceşti operatori nu au nici o legătură unul cu celălalt. Atât & cât şi * au prioritate faţă de toţi operatorii aritmetici cu excepţia lui minus unar, cu care au aceeaşi precedenţă.

Trebuie să vă asiguraţi că variabilele dvs. de tip pointer indică întotdeauna tipul corect de date. De exemplu, când declaraţi un pointer ca fiind de tipul **int**, compilatorul înţelege că adresa pe care o conţine memorează o variabilă de tip întreg - chiar dacă este adevărat sau nu. Deoarece C permite să atribuiţi orice adresă unei variabile de tip pointer, următorul fragment de cod este compilat corect, fără mesaje de eroare (sau doar avertismente, în funcţie de compilator), dar nu produce rezultatul dorit.

```
void main(void)
{
float x, y;
int *p;

/* Urmatoarea instructiune determina p (care este un
   pointer de tip intreg) sa indice catre un float. */
p = &x;

/* Urmatoarea instructiune nu lucreaza asa cum ne
   asteptam.*/
y = *p;
}
```

El nu va atribui valoarea din **x** lui **y**. Deoarece **p** este declarat ca un pointer de tip întreg, vor fi transferaţi în **y** doar doi octeţi din informaţie, nu toţi 8 care formează în mod normal un număr în virgulă mobilă.

NOTĂ: În C++ este interzisă convertirea unui pointer în altul fără utilizarea explicită a unui modelator de tip. Din acest motiv, programul precedent nici măcar nu va fi compilat dacă încercaţi să îl compilaţi ca pe un program în C++ (în loc de C). Totuşi, tipul de eroare descris poate să apară în C++ într-un mod mai indirect.

Expresii cu pointeri

În general, expresiile care implică pointeri se conformează aceloraşi reguli ca şi celelalte expresii. Această secţiune examinează câteva aspecte mai deosebite ale expresiilor care conţin pointeri.

Instrucţiuni de atribuire pentru pointeri

Ca şi pentru celelalte variabile, pentru a atribui valoarea din membrul drept al unei instrucţiuni de atribuire unui pointer, puteţi să utilizaţi un alt pointer. Iată un exemplu:

```
#include <stdio.h>
void main(void)
{
    int x;
    int *p1, *p2;

    p1 = &x;
    p2 = p1;

    printf(" %p", p2); /* afiseaza adresa lui x, nu
                          valoarea sa! */
}
```

Atât **p1** cât şi **p2** indică acum spre **x**. Adesa lui **x** este afişată utilizând specificatorul de formatare **printf() %p**, care face ca **printf()** să afişeze adresa în formatul utilizat de calculatorul pe care se lucrează.

Aritmetica pointerilor

Există doar două operaţii aritmetice care se pot efectua cu pointeri: adunarea şi scăderea. Pentru a înţelege ce se întâmplă în aritmetica pointerilor, să luăm un pointer de tip întreg **p1** cu valoarea efectivă 2000. Să mai presupunem că întregii au doi octeţi. După expresia:

```
p1++;
```

p1 va conţine 2002, nu 2001. Motivul este că de fiecare dată când **p1** este incrementat, el va indica spre următorul întreg. Acelaşi lucru este valabil şi pentru decrementare. De exemplu, presupunând că **p1** are valoarea 2000, expresia:

```
p1--;
```

determină ca **p1** să aibă valoarea 1998.

Generalizând exemplul precedent, aritmetica pointerilor este guvernată de următoarele reguli. De fiecare dată când este incrementat un pointer, el indică spre

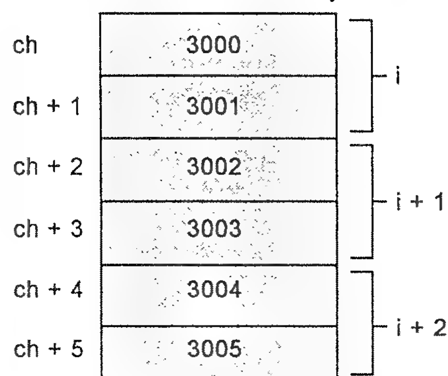


Figura 5-2 *Aritmetica pointerilor este relativă la tipul lor de bază*

ocația din memorie a următorului element de acelaşi tip cu tipul său de bază. De fiecare dată când este decrementat, el indică locația elementului anterior. Atunci când se aplică pointerilor de tip caracter, această aritmetică va fi una „normală" deoarece caracterele au întotdeauna lungimea de un octet. Însă, toți ceilalți pointeri vor fi incrementați şi decrementați cu lungimea tipului de date către care indică. Această caracteristică asigură că pointerul indică mereu spre un element de tipul său de bază. **Figura 5-2** ilustrează acest concept.

Nu sunteți limitați la operatorii de incrementare şi de decrementare. De exemplu, puteți aduna şi scădea întregi la sau din pointeri. Expresia:

```
p1 = p2 + 12;
```

face ca **p1** să indice al doisprezecelea element de acelaşi tip cu **p1** după cel pe care îl indică în mod curent.

În afara adunării şi a scăderii dintre un pointer şi un întreg, mai este permisă o singură operație aritmetică: puteți să scădeți un pointer din altul pentru a determina numărul de obiecte de acelaşi tip care separă cei doi pointeri. Toți ceilalți operatori aritmetici sunt interzişi. Nu puteți înmulți sau împărți pointeri; nu puteți să adunați doi pointeri; nu puteți să le aplicați operatorii pe biți; şi nu puteți aduna sau scădea tipul **float** sau **double** din pointeri.

Compararea pointerilor

Puteți să comparați doi pointeri într-o expresie relațională. De exemplu, fiind dați doi pointeri, **p** şi **q**, următoarea instrucțiune este perfect valabilă:

```
if(p<q) printf("p indica o memorie mai mica decit q\n)";
```

În general, compararea pointerilor este utilizată când doi sau mai mulți pointeri indică acelaşi obiect, aşa cum este o matrice. Ca exemplu, se creează o pereche de rutine stivă (stack) ca să stocheze şi să găsească valori întregi. O stivă este o listă care foloseşte accesul de tip „primul venit ultimul plecat". Ea este des comparată cu o stivă de farfurii de pe o masă - prima aşezată este ultima folosită. Memoriile stivă sunt des utilizate în compilatoare, interpretoare, foi de calcul şi alte sisteme înrudite acestora. Pentru a crea o stivă vă sunt necesare două funcții: **pune()** şi **scoate()**. Funcția **pune()** plasează valori în stivă iar **scoate()** le preia. Aceste rutine sunt prezentate aici împreună cu o simplă funcție **main()** care le conduce. Programul aşează în stivă valorile pe care le introduceți. Dacă introduceți **0**, iese o valoare din stivă. Pentru a opri programul, introduceți **-1**.

```
#include <stdio.h>
#include <stdlib.h>

#define MARIME 50

void pune(int i);
int scoate(void);

int *vis, *pl, stiva[MARIME];

void main(void)
{
    int valoare;
    vis = stack; /* vis indica virful stivei */
    pl = stack;  /* initializeaza pl */

    do {
        printf("Introduceti valoare: ");
        scanf("%d", &valoare);
        if(valoare!=0) pune(valoare);
        else printf("valoarea din virf este %d\n",
                    scoate());
```

```
    }     while(valoare!=-1);
}

void pune(int i)
{
    p1++;
    if(p1==(vis+MARIME)) {
          printf("Stiva supraincarcata");
          exit(1);
    }
    *p = i;
}

scoate(void)
{
    if(p1==vis) {
          printf("Stiva vida");
          exit(1);
    }
    p1--;
    return *(p1+1);
}
```

Puteți vedea că memoria pentru stivă este asigurată de matricea **stack**. Pointerul **p1** este inițializat pentru a indica primul octet din **stack**. Cu ajutorul variabilei **p1** aveți efectiv acces la stivă. Variabila **vis** păstrează adresa din memorie a vârfului stivei. Valoarea din **vis** împiedică depășirea superioară sau inferioară a stivei. Odată inițializată stiva, pot fi folosite **pune()** și **scoate()**. Atât **pune()** cât și **scoate()** efectuează un test relațional asupra pointerului **p1** pentru a găsi erorile de limite. În **pune()**, **p1** este comparat cu sfârșitul stivei adunându-se la **vis** și **MARIME** (mărimea stivei). Aceasta previne o supraîncărcare. În **scoate()**, **p1** este comparat cu **vis** pentru a se asigura că nu a apărut o depășire inferioară.

În **scoate()** sunt necesare paranteze în instrucțiunea **return**. Fără ele, ea ar arăta astfel:

```
return *p1 + 1;
```

În această formă, instrucțiunea ar returna valoarea locației **p1** plus unu, nu valoarea locației **p1+1**.

Pointeri și matrice

Există o strânsă legătură între pointeri și matrice. Să luăm acest fragment de program:


```
char sir[80], *p1;
p1 = sir;
```

Aici, **p1** a fost inițializat cu adresa primului element din **sir**. Pentru a avea acces la al cincilea element din **sir**, puteți să scrieți:

```
sir[4]
```

sau

```
*(p1+4)
```

Ambele instrucțiuni vor returna al cincilea element. Amintiți-vă că matricele încep de la 0. Pentru a avea acces la al cincilea element, trebuie să folosiți indicele 4 pentru **sir**. De asemenea, adăugați 4 la pointerul **p1** pentru a avea acces la al cincilea element, deoarece **p1** indică efectiv spre primul element din **sir**. (Amintiți-vă că un nume de matrice fără indice returnează adresa de pornire a acelei matrice, care este adresa primului ei element.)

C furnizează două metode de acces la elementele matricei: aritmetica pointerilor și indicii matricelor. Prima poate fi mai rapidă decât cealaltă. De vreme ce viteza este deseori un criteriu în programare, programatorii în C/C++ utilizează de obicei pointeri pentru a avea acces la elementele matricei.

Următoarele două versiuni ale lui **scriesir()** - una cu indici de matrice și una cu pointeri - ilustrează cum puteți să utilizați pointerii în loc indici pentru matrice. Funcția **scriesir()** scrie un șir la dispozitivul de ieșire standard, câte un caracter o dată.

```
/* Se utilizeaza s ca matrice cu indecsi. */
void scriesir(char *s)
{
    register int t;

    for(t=0; s[t]; ++t) putchar(s[t]);
}
/* Se utilizeaza s ca pointer. */
void scriesir(char *s)
{
    while(*s) putchar(*s++);
}
```

Programatorilor profesioniști în C/C++, în marea lor majoritate, vor găsi a doua variantă mai ușor de citit și de înțeles. De fapt, varianta cu pointer este modul comun de scriere a rutinelor de acest fel în C/C++.

Matrice de pointeri

Pointerii pot fi organizaţi în matrice ca oricare alt tip de date. Declararea unei matrice de pointeri de tipul **int**, cu mărimea 10 este:

```
int *x[10];
```

Pentru a atribui adresa unei variabile de tip întreg cu numele **var** elementului al treilea al matricei de pointeri, scrieţi:

```
x[2] = &var;
```

Pentru a obţine valoarea **var**, scrieţi:

```
*x[2]
```

Dacă doriţi să transmiteţi o matrice de pointeri unei funcţii, puteţi folosi aceeaşi metodă pe care o folosiţi pentru a transmite alte matrice - apelaţi pur şi simplu funcţia cu numele matricei fără nici un indice. De exemplu, o funcţie care primeşte matricea **x** arată astfel:

```
void afis_matrice(int *q[])
{
    int t;

    for(t=0; t<10; t++)
          printf("%d ", *q[t]);
}
```

Reţineţi că **q** nu este un pointer către întregi, ci un pointer către o matrice de pointeri către întregi. De aceea, trebuie să declaraţi parametrul **q** ca o matrice de pointeri pentru întregi, aşa cum tocmai am arătat. Nu puteţi să îl declaraţi pe **q** ca simplu pointer pentru întregi deoarece aceasta este cu totul altceva.

Matricele de pointeri sunt deseori folosite pentru a păstra pointeri către şiruri. Puteţi crea o funcţie care emite un mesaj de eroare prin numele său de cod, aşa cum se prezintă aici:

```
void eroare_sintaxa(int num)
{
    static char *err[] = {
          "Nu pot sa deschid fisierul\n",
          "Eroare de citire\n",
          "Eroare de scriere\n",
          "Suport magnetic defect\n"
    };

    printf("%s", err[num]);
}
```

Matricea **err** păstrează pointeri către fiecare şir. După cum puteţi vedea, **printf()** din interiorul lui **eroare_sintaxa()** este apelată cu un pointer de tip caracter care indică spre unul dintre mesajele de eroare corespunzător numărului acelei erori transmise funcţiei. De exemplu, dacă lui **num** îi este transmis 2, va fi afişat mesajul **Eroare de scriere**.

Este interesant să notaţi că linia de comandă a argumentului **argv** este o matrice de pointeri de tip caracter (a se vedea **Capitolul 6**).

Indirectare multiplă

Puteţi să aveţi un pointer care să indice pe un altul care indică valoarea ţintă. Această situaţie este denumită *indirectare multiplă* sau *pointeri către pointeri.* Pointerii către pointeri pot produce confuzii. **Figura 5-3** ajută la clarificarea conceptului de indirectare multiplă. După cum puteţi vedea, valoarea unui pointer obişnuit este adresa obiectului care conţine valoarea dorită. În cazul unui pointer către un pointer, primul conţine adresa celui de-al doilea pointer, care indică spre obiectul care conţine valoarea dorită.

Indirectarea multiplă poate fi continuată cât de mult doriţi, dar mai mult decât un pointer către un pointer este rareori necesar. De altfel, indirectarea excesivă este derutantă şi sursă de erori conceptuale.

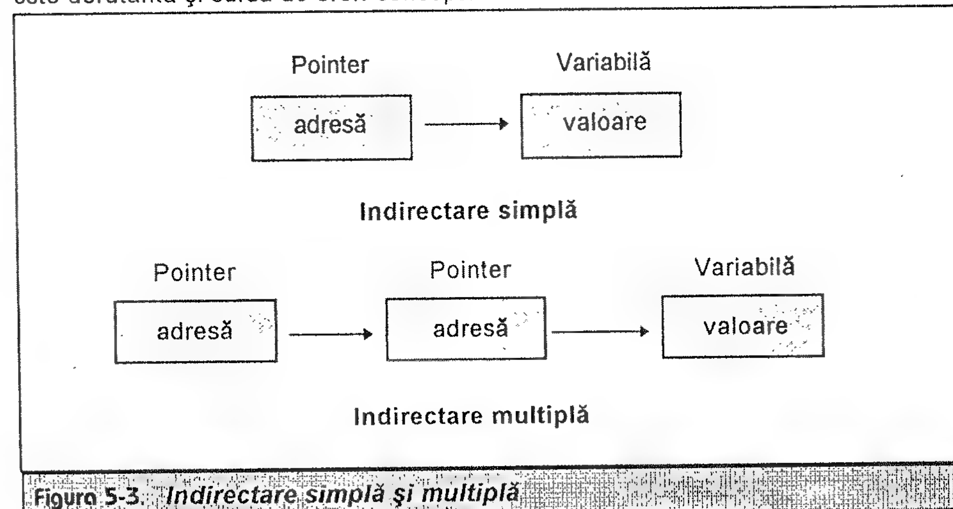


Figura 5-3. *Indirectare simplă şi multiplă*



NOTĂ: Nu confundaţi indirectarea multiplă cu structurile de date de nivel înalt, cum ar fi listele înlănţuite, care conţin pointeri. Cele două concepte se deosebesc fundamental.

O variabilă care este pointer către pointer se declară ca atare, plasându-se încă un asterisc în faţa numelui. De exemplu, următoarea declarare spune compilatorului că **noulbilant** este un pointer către un pointer de tipul **float**.

```
float **noulbilant;
```

Trebuie să înţelegeţi că **noulbilant** nu este un pointer către un număr în virgulă mobilă ci un pointer către un pointer de tipul **float**.

Pentru a avea acces la valoarea dorită indicată de un pointer la un pointer, trebuie să aplicaţi de două ori operatorul asterisc, ca în următorul exemplu:

```
#include <stdio.h>

void main(void)
{
    int x, *p, **q;

    x = 10;
    p = &x;
    q = &p;
    printf("%d", **q); /* afiseaza valoarea lui x */
}
```

Aici, **p** este declarat ca un pointer către un întreg iar **q** ca un pointer către un pointer către un întreg. Apelarea funcţiei **printf()** afişează pe ecran numărul 10.

Iniţializarea pointerilor

După ce este declarat un pointer, dar înainte de a i se atribui o valoare, el poate să conţină o valoare necunoscută.



ATENŢIE: Dacă veţi încerca să folosiţi un pointer înainte de a-i da o valoare validă, probabil că vă veţi bloca programul - eventual şi sistemul de operare al calculatorului dvs. - un tip de eroare foarte neplăcută!

Există o convenţie importantă pe care o respectă majoritatea programatorilor în C/C++ atunci când lucrează cu pointeri: unui pointer care nu indică efectiv o locaţie de memorie validă i se dă valoarea null (care este zero). Prin convenţie, orice pointer care este null nu indică spre nimic şi nu ar trebui folosit. Totuşi, doar faptul că un pointer are valoarea null nu îl face „sigur". Utilizarea lui null este o simplă convenţie pe care o urmează programatorii. Ea nu face parte din limbajul C/C++. De exemplu, dacă utilizaţi un pointer null în membrul stâng al unei instrucţiuni de atribuire, încă mai există riscul de a bloca programul sau sistemului de operare.

Deoarece un pointer null se presupune că este nefolosit, puteţi să îl utilizaţi pentru a face multe dintre rutinele cu pointeri mai uşor de codificat şi mai eficiente. De exemplu, puteţi folosi un pointer null pentru a marca sfârşitul unei matrice de pointeri. O rutină care are acces la acea matrice ştie că a ajuns la sfârşit când întâlneşte valoarea null. Funcţia **cauta()** prezentată aici ilustrează această facilitate.


```
/* cauta un nume */
cauta(char *p[], char *nume)
{
    register int i;

    for(t=0; p[t]; ++t)
        if(!strcmp(p[t], nume)) return t;

        return -1; /* nu am gasit */
}
```

Bucla **for** din interiorul lui **cauta()** rulează până când găseşte şirul pe care îl caută sau întâlneşte un pointer null. Presupunând că sfârşitul unei matrice este marcat printr-un null, condiţia de control a buclei devine falsă când se ajunge la el.

Programatorii de C/C++ iniţializează de obicei şirurile. Aţi văzut un asemenea exemplu în funcţia **eroare_sintaxa()** din secţiunea „Matrice de pointeri". O altă variaţiune pe tema iniţializării este următorul tip de declarare de şir:

```
char *p = "salut lume";
```

După cum puteţi vedea, pointerul **p** nu este o matrice. Motivul pentru care acest tip de iniţializare merge se datorează modului de funcţionare a compilatorului. Toate compilatoarele de C/C++ creează ceea ce se numeşte *tabel de şiruri*, care este utilizat de calculator pentru a memora constantele de tip şir pe care le foloseşte programul. De aceea, instrucţiunea precedentă de declarare plasează adresa lui **salut lume**, aşa cum este ea stocată în tabela de şiruri, în pointerul **p**. De-a lungul programului **p** poate fi utilizat ca oricare alt şir. De exemplu, următorul program este perfect valabil:

```
#include <stdio.h>
#include <string.h>

char *p = "salut lume";

void main(void)
{
    register int t;

    /* afiseaza sirul inainte si inapoi */
    printf(p);
    for(t=strlen(p)-1; t>-1;t--) printf("%c", p[t]);
}
```

Pointeri către funcţii

O caracteristică generatoare de confuzii dar performantă a limbajului C este *pointerul către o funcţie*. Chiar dacă o funcţie nu este o variabilă, ea are o localizare în memorie care poate fi atribuită unui pointer. Adresa unei funcţii este punctul de intrare în funcţie. Din această cauză un pointer către funcţie poate fi utilizat pentru a apela o funcţie.

Pentru a înţelege cum lucrează aceasta, trebuie să cunoaşteţi puţin modul în care este compilată şi apelată o funcţie. Mai întâi, pe măsură ce este compilată fiecare funcţie, codul sursă este transformat în cod obiect şi se stabileşte un punct de intrare. În timpul rulării programului, atunci când este apelată o funcţie, acest punct de inserare este apelat de limbajul maşină. De aceea, dacă un pointer conţine adresa punctului de intrare, poate fi folosit pentru a apela acea funcţie.

Adresa unei funcţii se obţine utilizând numele funcţiei fără nici o paranteză sau argumente (similar cu modul în care se obţine adresa unei matrice când este utilizat doar numele ei, fără indici). Pentru a vedea cum se face aceasta, urmăriţi următorul program, fiind atenţi la declaraţii.

```
#include <stdio.h>
#include <string.h>

void cauta(char *a, char *b,
        int (*comp)(const char *, const char *));

void main(void)
{
    char s1[80], s2[80];
    int (*p)(const char *, const char *);
```



```
    p = strcmp;

    gets(s1);
    gets(s2);
    cauta(s1, s2, p);
}

void cauta(char *a, char *b,
        int (*comp)(const char *, const char *))
{
    printf("testeaza egalitatea\n");
    if(!(*comp)(a, b)) printf("egal");
    else printf("diferit");
}
```

Când este apelată funcţia **cauta()**, sunt transmişi ca parametri doi pointeri de tip caracter şi unul către o funcţie. În interiorul funcţiei **cauta()** argumentele sunt declarate ca pointeri de tip caracter şi pointer către o funcţie. Reţineţi modul în care este declarat pointerul către funcţie. Trebuie să folosiţi o formă similară când declaraţi alţi pointeri către funcţii, chiar dacă tipul returnat şi parametrii funcţiei pot să difere. Parantezele pentru ***comp** sunt necesare pentru compilator pentru a interpreta corect această instrucţiune.

Expresia:

```
(*comp) (a, b)
```

din interiorul lui **cauta()** apelează cu argumentele **a** şi **b** pe **strcmp()**, care este indicată de **comp**. Repetăm, parantezele pentru ***comp** sunt necesare. Acest exemplu ilustrează, de asemenea, metoda generală pentru utilizarea unui pointer către o funcţie, pentru a apela funcţia pe care o indică.

Reţineţi că puteţi apela **cauta()** folosind direct **strcmp()**, aşa cum se prezintă aici:

```
cauta(s1, s2, strcmp);
```

Aceasta elimină cerinţa unei variabile în plus, de tip pointer.

Vă puteţi întreba de ce ar dori cineva să scrie un program în acest mod. Evident, în primul exemplu nu se câştigă nimic şi se introduce un grad semnificativ de confuzie. Totuşi, din când în când este avantajos să transmiteţi funcţii ca parametri sau să creaţi o matrice de funcţii. De exemplu, când se scrie un compilator sau un interpretor, modulul parser (partea care evaluează expresia) apelează deseori diverse funcţii de completare, cum sunt acelea care efectuează operaţiile matematice (sin, cos, tg ş.a.m.d.), care asigură operaţiile I/O sau care

oferă accesul la resursele sistemului. În locul unei mari instrucțiuni **switch**, cu toate aceste funcții listate în ea, poate fi creată o matrice de pointeri pentru funcții. În acest mod, este selectată prin indici funcția adecvată. Puteți avea imaginea acestui tip de utilizare studiind versiunea extinsă a exemplului precedent. În acest program, **cauta()** poate să caute egalitatea alfabetică sau cea numerică prin simpla apelare cu diverse funcții de comparare.

```
#include <stdio.h>
#include <ctype.h>
#include <stdlib.h>
#include <string.h>

void cauta(char *a, char*b,
           int (*comp)(const char *, const char *));
int numcomp(const char *a, const char *b);

void main(void)
{
   char s1[80], s2[80];

   gets(s1);
   gets(s2);

   if(isalfa(*s1))
         cauta(s1, s2, sircomp);
   else
         cauta(s1, s2, numcomp);
}

void cauta(char *a, char *b,
           int (*comp)(const char *, const char *))
{
   printf("testeaza egalitatea\n");
   if(!(*comp)(a, b)) printf("egal");
   else printf("diferit");
}

numcomp(const char *a, const char *b)
{
   if(atoi(a)==atoi(b)) return 0;
   else return 1;
}
```


Funcții de alocare dinamică în C

Pointerii oferă suportul necesar pentru sistemul puternic de alocare dinamică în C. *Alocarea dinamică* este caracteristica prin care un program poate obține memorie în timpul rulării. După cum știți, variabilelor globale li se alocă memorie în timpul compilării. Variabilele locale folosesc memoria stivă. Totuși, nici variabilele globale nici cele locale nu pot fi adăugate în timpul execuției programului. Vor fi momente însă în care memoria necesară programului nu poate fi cunoscută dinainte. De exemplu, un procesor de texte sau o bază de date pot să beneficieze de întreaga memorie RAM a sistemului. Însă cantitatea de RAM diferă între calculatoare, astfel încât programele nu vor putea face aceasta folosind variabile obișnuite. Ele trebuie să aloce memorie după necesități, folosind sistemul de alocare dinamică existent în C.



***NOTĂ:** Chiar dacă C++ acceptă pe deplin sistemul de alocare dinamică al lui C, el își definește propriul sistem, care conține mai multe îmbunătățiri față de cele din C. Sistemul de alocare dinamică al lui C++ este discutat în Partea a doua.*

Memoria alocată de funcțiile de alocare dinamică în C este obținută din *heap* - regiunea de memorie liberă care există între zona permanentă de memorie a programului dvs. și cea stivă. Chiar dacă mărimea zonei heap este necunoscută, ea conține, în general, o cantitate destul de mare de memorie liberă.

Nucleul sistemului de alocare din C constă din funcțiile **malloc()** și **free()**. (Majoritatea compilatoarelor asigură și alte funcții de alocare dinamică, dar acestea două sunt cele mai importante.) Aceste funcții lucrează în pereche folosind zona de memorie liberă pentru a stabili și a păstra o listă cu memoria disponibilă. Funcția **malloc()** alocă memorie iar **free()** o eliberează. Aceasta înseamnă că de fiecare dată când îi este cerută memorie lui **malloc()**, se alocă o zonă din memoria rămasă liberă. De fiecare dată când este apelată funcția **free()**, memoria este returnată sistemului. Orice program care folosește aceste funcții trebuie să includă fișierul antet STDLIB.H.

Funcția **malloc()** are acest prototip:

void *malloc(size_t *numar_de_octeți*);

Aici, *număr_de_octeți* este numărul de octeți din memorie pe care doriți să îl alocați. (Tipul **size_t** este definit în STDLIB.H aproximativ ca un întreg de tip **unsigned**.) Funcția **malloc()** returnează un pointer de tipul **void**, ceea ce înseamnă că îl puteți atribui oricărui tip de pointer. După o apelare reușită, **malloc()** returnează un pointer spre primul octet al regiunii de memorie alocate în memoria liberă. Dacă nu există suficientă memorie disponibilă pentru a satisface cerința lui **malloc()**, apare o blocare de alocare iar **malloc()** returnează null.

Fragmentul de cod prezentat aici alocă 1000 de octeți de memorie contiguă:

```
char *p;
p = malloc(1000); /* preia 1000 octeti */
```

După alocare, **p** indică spre primul din cei 1000 de octeți de memorie liberă.

Observați că nu este necesar nici un modelator de tip pentru a atribui lui **p** valoarea returnată de **malloc()**. În C, un pointer de tip ***void** este convertit automat în tipul pointerului din partea stângă a instrucțiunii de atribuire. Totuși, este important de reținut că această conversie automată *nu are loc* în C++. Mai mult, în C++ este necesar un modelator explicit de tip când este atribuit un pointer de tip ***void** unui alt tip de pointer. De aceea, în C++, atribuirea precedentă trebuie să fie scrisă astfel:

```
p = (char *) malloc(1000);
```

Ca regulă generală, în C++ trebuie să folosiți un modelator de tip când atribuiți (sau altfel spus, convertiți) un tip de pointer într-altul. Aceasta este una dintre puținele diferențe fundamentale între C și C++.

Următorul exemplu alocă spațiu pentru 50 de întregi. Rețineți utilizarea lui **sizeof** pentru asigurarea portabilității.

```
int *p;
p = malloc(50*sizeof(int));
```

Deoarece memoria nu este infinită, când alocați memorie trebuie să verificați valoarea returnată de **malloc()** înainte de a folosi pointerul, pentru a vă asigura că nu este null. Utilizând un pointer null sigur veți bloca programul. Modul corect de alocare a memoriei și de testare a validității unui pointer este ilustrat în acest fragment de cod:

```
if(!(p=malloc(100))) {
    printf("Depasire de memorie.\n");
    exit(1);
}
```

Desigur, puteți înlocui **exit()** cu un program de tratare a erorilor. Doar asigurați-vă că nu veți folosi pointerul **p** dacă este null.

Funcția **free()** este opusă lui **malloc()** deoarece ea returnează în sistem memoria alocată anterior. O dată memoria eliberată, ea poate să fie refolosită de o apelare ulterioară a lui **malloc()**. Funcția **free()** are următorul prototip:

void free(void *p*);

Aici, *p* este un pointer spre memoria care a fost alocată anterior folosind **malloc()**. Este esențial să nu apelați *niciodată* **free()** cu un argument impropriu; aceasta v-ar distruge lista de memorie liberă.

Probleme ale pointerilor

Nimic nu vă va crea mai multe probleme decât un pointer greșit! Pointerii sunt în același timp blagosloviți și blestemați. Ei vă oferă o putere de temut și sunt necesari multor programe. În același timp, când un pointer conține accidental o valoare greșită, poate fi greșeala cea mai greu de depistat.

Un pointer eronat este dificil de găsit deoarece nu pointerul însuși constituie problema. Necazul este că de fiecare dată când efectuați o operație folosind pointerul greșit, citiți sau scrieți într-o zonă de memorie necunoscută. Dacă citiți din ea, cel mai rău lucru care se poate întâmpla este să obțineți valori greșite. Însă, dacă scrieți în ea, poate că o faceți peste alte zone ale codului sau ale datelor dvs. Aceasta nu se va vedea decât mai târziu, în timpul execuției programului, și vă poate face să căutați greșeala în altă parte. Nu există aproape nici o dovadă care să sugereze că pointerul este cauza inițială a problemei. Acest tip de greșeală determină ca programatorii să piardă zilele și nopțile. Deoarece erorile pointerilor sunt asemenea coșmaruri, ar fi bine să faceți totul pentru a nu genera vreuna. Pentru a vă ajuta să le evitați, vom discuta aici câteva din erorile uzuale.

Exemplul clasic de greșeală pentru pointeri este *pointerul neinițializat*. Să luăm următorul program:

```
/* Acest program este gresit. */
void main(void)
{
    int x, *p;

    x = 10;
    *p = x;
}
```

Acest program atribuie valoarea 10 unei locații de memorie necunoscute. Iată de ce. De vreme ce pointerului **p** nu i s-a dat o valoare, el conține una necunoscută atunci când are loc atribuirea ***p = x**, ceea ce face ca valoarea din **x** să fie scrisă într-o locație de memorie necunoscută. Acest tip de problemă scapă de obicei neobservată când programul este mic deoarece cele mai mari șanse sunt ca **p** să conțină o adresă „sigură" - una care nu intră în zona de program, de date ori a sistemului de operare. Dar, pe măsură ce programul dvs. se mărește, crește și probabilitatea ca **p** să conțină ceva vital. În cele din urmă, programul se va opri. Soluția este să vă asigurați mereu, înainte de a utiliza un pointer, că acesta indică spre ceva valid.

O a doua eroare curentă este determinată de simpla neînţelegere a modului de folosire a unui pointer. Să luăm următorul cod:

```
/* Acest program este gresit. */
#include <stdio.h>

void main(void)
{
    int x, *p;

    x = 10;
    p = x;

    printf("%d", *p);
}
```

Apelarea funcţiei **printf()** nu afişează pe ecran valoarea lui **x**, care este 10. Ea afişează o valoare necunoscută datorită atribuirii greşite:

```
p = x;
```

Această instrucţiune atribuie valoarea 10 pointerului **p**. Totuşi **p** se presupune că memorează o adresă şi nu o valoare. Pentru a corecta programul, scrieţi:

```
p = &x;
```

O altă eroare care apare uneori este determinată de presupunerea incorectă asupra amplasării variabilelor în memorie. Nu puteţi şti niciodată unde vor fi plasate în memorie datele dvs., sau dacă vor fi din nou plasate în acelaşi mod ori dacă fiecare compilator le va trata în acelaşi fel. Din acest motiv, comparaţiile între pointeri care nu indică acelaşi obiect pot să determine rezultate neaşteptate. De exemplu,

```
char s[80], y[80];
char *p1, *p2;

p1 = s;
p2 = y;
if(p1 < p2)...
```

este în general un concept greşit. (În cazuri deosebite, puteţi să folosiţi aşa ceva pentru a determina poziţia relativă a variabilelor. Dar aceasta se întâmplă rar.)

O eroare asemănătoare rezultă când presupuneţi că două matrice alăturate pot fi indexate ca una singură prin simpla incrementare a pointerului peste graniţele matricelor, aşa cum se prezintă aici:

```
int prima[10], adoua[10];
int *p, t;

p = prima;
for(t=0; t<20; ++t) *p++ = t;
```

Aceasta nu este o cale corectă de iniţializare a matricelor **prima** şi **adoua** cu numere de la **0** la **19**. Chiar dacă poate să lucreze pe anumite compilatoare şi în anumite condiţii, ea presupune că ambele matrice vor fi plasate una lângă alta în memorie, **prima** fiind prima. Nu va fi întotdeauna cazul.

Următorul program ilustrează un tip de greşeală foarte periculoasă. Încercaţi să o descoperiţi.

```
/* Acest program contine o greseala. */
#include <string.h>
#include <stdio.h>

void main(void)
{
    char *p1;
    char s[80];

    p1 = s;
    do    {
          gets(s); /* citeste sir */
          /* afiseaza echivalentul in notatie zecimala al
             fiecarui caracter */
          while(*p1) printf(" %d", *p1++);
    }     while(strcmp(s, "gata"));
}
```

Acest program foloseşte **p1** pentru a afişa valorile ASCII asociate caracterelor conţinute în **s**. Problema este că lui **p1** îi este atribuită adresa lui **s** o singură dată. Prima dată în buclă, **p1** indică primul caracter din **s**. Însă a doua oară el continuă de unde a rămas deoarece nu este reiniţializat cu începutul lui **s**. Acest caracter care urmează poate face parte din al doilea şir, din altă variabilă sau poate fi o parte din program! Modul corect de scriere a programului este arătat aici:

```
/* Acest program este acum corect. */
#include <string.h>
#include <stdio.h>

void main(void)
{
    char *p1;
    char s[80];

    do    {
          p1 = s;
          gets(s); /* citeste sir */
          /* afiseaza echivalentul in notatie zecimala al
             fiecarui caracter */
          while(*p1) printf(" %d", *p1++);

    }     while(strcmp(s, "gata"));
}
```

Aici, la fiecare iterare a buclei, **p1** este iniţializat la începutul şirului. În general, trebuie să vă amintiţi să reiniţializaţi un pointer când îl refolosiţi.

Faptul că manevrarea incorectă a pointerilor poate să genereze greşeli subtile, nu este un motiv să nu îi folosiţi. Doar să fiţi atenţi şi să fiţi siguri că ştiţi spre ce indică fiecare pointer înainte de a-l utiliza.

Capitolul 6

Funcţii



Funcţiile sunt construcţii bloc în C şi locul în care se petrece întreaga activitate a programului. Ele sunt una din cele mai importante caracteristici ale limbajului C.

Forma generală a unei funcţii

Forma generală a unei funcţii este:

```
specificator_de_tip nume_funcţie(listă_de_parametri)
{
      corpul funcţiei
}
```

Specificatorul_de_tip specifică tipul de date pe care îl returnează funcţia. O funcţie poate returna orice tip de date cu excepţia unei matrice. Dacă nu este specificat nici un tip, compilatorul presupune că acea funcţie returnează un rezultat de tip întreg. *Lista_de_parametri* este o listă separată prin virgule de nume de variabile şi tipurile lor asociate care primesc valorile argumentelor atunci când este apelată funcţia. O funcţie poate să nu aibă parametri, caz în care lista lor este vidă. Totuşi, parantezele sunt necesare chiar dacă nu există parametri.

În declaraţiile de variabile puteţi să vă referiţi la mai multe variabile de acelaşi tip folosind o listă cu numele lor separate prin virgulă. Spre deosebire de acestea, toţi parametrii funcţiei trebuie declaraţi individual, fiecare conţinând atât tipul cât şi numele. Aceasta înseamnă că lista de declarare a parametrilor unei funcţii are forma generală:

f(tip numevariab1, tip numevariab2,..., tip numevariabN)

Iată, de exemplu, declaraţii corecte şi incorecte de parametri pentru funcţii:

```
f(int i, int k, int j) /* corect */
f(int i, k, float j)   /* incorect */
```

Sfera de influenţă a funcţiilor

Sfera de influenţă a unui limbaj este formată din regulile care stabilesc ce secvenţă de cod ştie sau are acces la o altă secvenţă de cod sau de date.

Fiecare funcţie este un bloc de cod discret. Codul unei funcţii este propriu ei şi nici o instrucţiune din altă funcţie nu poate să aibă acces la el decât printr-un apel al funcţiei. (De exemplu, nu puteţi folosi **goto** pentru a sări în mijlocul altei funcţii.) Codul care constituie corpul unei funcţii este ascuns de restul programului şi, dacă nu utilizează variabile sau date globale, nu poate fi afectat şi nu poate afecta alte părţi ale programului. Cu alte cuvinte, codul şi datele care sunt definite într-o funcţie nu pot să interacţioneze cu codul sau cu datele definite în alta, deoarece cele două funcţii nu au aceeaşi sferă de influenţă.

Variabilele care sunt definite într-o funcţie sunt numite *variabile locale*. O variabilă locală este creată atunci când se intră în acea funcţie şi este distrusă la ieşire. Aceasta înseamnă că variabilele locale nu îşi păstrează valoarea între apelările funcţiei. Singura excepţie de la această regulă este atunci când variabila este declarată cu specificatorul de clasă de memorare **static**. Aceasta determină compilatorul să trateze variabila ca şi cum ar fi o variabilă globală, în scopul stocării ei, dar încă îi limitează sfera la interiorul funcţiei. (**Capitolul 2** studiază în amănunt variabilele globale şi locale.)

În C (şi C++), toate funcţiile au acelaşi nivel al sferei de influenţă. Aceasta înseamnă că nu puteţi defini o funcţie într-o funcţie, motiv pentru care nici C şi nici C++ nu sunt, tehnic vorbind, limbaje structurate în blocuri.

Argumentele funcţiei

Dacă o funcţie urmează să folosească argumente, ea trebuie să declare variabile care acceptă valori ale argumentelor. Aceste variabile sunt denumite *parametri formali* ai funcţiei. Ei se comportă ca şi celelalte variabile locale din funcţie şi sunt create la intrarea într-o funcţie şi distruse la ieşirea din ea. Aşa cum se arată în următoarea funcţie, declararea parametrilor are loc după numele funcţiei.

```
/* Returneaza 1 daca c face parte din sirul s; altfel, 0. */
este_in(char *s, char c)
{
   while(*s)
         if(*s==c) return 1;
         else s++;
   return 0;
}
```

Funcţia **este_in()** are doi parametri: **s** şi **c**. Această funcţie returnează 1 atunci când caracterul **c** face parte din şirul **s**; altfel, ea returnează 0.

Ca şi pentru variabilele locale, parametrilor formali ai funcţiei puteţi să le aplicaţi atribuiri sau să îi folosiţi în orice expresie permisă. Chiar dacă aceste variabile îndeplinesc sarcina specială de primire a valorilor argumentelor transmise funcţiei, puteţi să le utilizaţi aşa cum o faceţi cu oricare alte variabile locale.

Apelare prin valoare, apelare prin referinţă

În general, în subrutine pot fi pasate argumente în unul din două feluri. Primul este denumit *apel prin valoare*. Această metodă copiază *valoarea* unui argument într-un

parametru formal al subrutinei. În acest caz, modificările efectuate asupra parametrului nu au efect asupra argumentului.

Al doilea mod de transfer de argumente către subrutine este *apelarea prin referință*. Prin această metodă, în parametru este copiată *adresa* unui argument. În interiorul subrutinei adresa este folosită pentru acces la argumentul folosit efectiv la apelare. Aceasta înseamnă că modificările efectuate asupra parametrului afectează argumentul.

Cu puține excepții, C folosește apelarea prin valoare pentru a transmite argumentul. În general, aceasta înseamnă că blocul de cod din interiorul funcției nu poate să modifice argumentele folosite pentru a apela funcția. Să luăm următorul program:

```
#include <stdio.h>

int patrat(int x);

void main(void)
{
    int t=10;

    printf("%d %d", patrat(t), t);
}

patrat(int x)
{
    x = x*x;
    return(x);
}
```

În acest exemplu, valoarea argumentului funcției **patrat()**, 10, este copiată în parametrul **x**. Când are loc atribuirea **x = x*x**, este modificată doar variabila locală **x**. Variabila **t**, folosită pentru a apela **patrat()**, are în continuare valoarea 10. De aceea, ieșirea este **100 10**.



REȚINEȚI: *Funcției îi este transmisă o copie a valorii argumentului. Ceea ce se întâmplă în interiorul funcției nu are efect asupra variabilei folosite pentru apelare.*

Crearea unei apelări prin referință

Chiar dacă prin convenție în C parametrul se transformă prin valoare, puteți să creați o apelare prin referință prin transmiterea unui pointer către argument, în loc de argumentul însuși. Dacă se transmite adresa argumentului, codul funcției poate modifica valoarea argumentului din afara sa.

Pointerii sunt transmiși funcțiilor la fel ca oricare altă valoare. Desigur, trebuie să declarați parametrii ca fiind de tip pointer. De exemplu, funcția **invers()**, care inversează între ele valorile a două variabile de tip întreg indicate de către argumentele sale, arată cum se face aceasta.

```
void modific(int *x, int *y)
{
    int temp;
    temp = *x; /* salveaza valoarea de la adresa x */
    *x = *y;      /* pune pe y in x */
    *y = temp; /* pune pe x in y */
}
```

invers() este capabilă să schimbe între ele valorile celor două variabile indicate de **x** și **y** deoarece sunt pasate adresele lor (nu valorile). Astfel, în cadrul funcției, putem avea acces la conținutul variabilelor utilizând operațiile standard pentru pointeri și putem schimba între ele valorile variabilelor folosite pentru apelarea funcției.

Rețineți că **invers()**, sau oricare altă funcție care utilizează pointeri ca parametri, trebuie să fie apelată cu *adresa argumentelor*. Următorul program arată modul corect de apelare a funcției **invers()**.

```
void modific(int *x, int *y);

void main(void)
{
    int i, j;

    i = 10;
    j = 20;
    modific(&i, &j); /* paseaza adresele lui i si j */
}
```

În acest exemplu, variabilei **i** îi este atribuită valoarea 10 iar lui **j** valoarea 20. Funcția **invers()** este apelată cu adresele lui **i** și **j**. (Operatorul unar & este folosit pentru a prelua adresele variabilelor.) Astfel, în funcția **invers()** sunt pasate adresele variabilelor **i** și **j**, nu valorile lor.

Apelarea funcțiilor cu matrice

Matricele sunt descrise în detaliu în Capitolul 4. Acest paragraf discută transmiterea matricelor ca argumente pentru funcții deoarece ele sunt o excepție

de la regula de transfer prin valoare a parametrilor.

Când o matrice este folosită ca un argument al unei funcții, acesteia îi este pasată adresa matricei. Astfel, codul funcției operează asupra conținutului efectiv al matricei folosite pentru apelarea funcției și poate să îl modifice. De exemplu, să considerăm funcția **afis_maj()**, care afişează un argument de tip şir cu majuscule.

```
#include <stdio.h>
#include <ctype.h>

void afis_maj(char *sir);

void main(void)
{
    char s[80];

    gets(s);
    afis_maj(s);
}

/* Afiseaza un sir cu majuscule. */
void afis_maj(char *sir)
{
    register int t;

    for(t=0; sir[t]; ++t) {
          sir[t] = toupper(sir[t]);
          putchar(sir[t]);
    }
}
```

După apelarea funcției **afis_maj**, conținutul matricei **s** din **main()** s-a modificat şi este alcătuit din majuscule. Dacă nu asta v-ați dorit, puteți scrie programul astfel:

```
#include <stdio.h>
#include <ctype.h>

void afis_maj(char *sir);

void main(void)
{
    char s[80];

    gets(s);
    afis_maj(s);
}

void afis_maj(char *sir)
{
    register int t;

    for(t=0; sir[t]; ++t)
          putchar(toupper(sir[t]));
}
```

În această versiune conținutul matricei **s** rămâne nemodificat deoarece valorile sale nu s-au schimbat.

Funcția din biblioteca standard **gets()** este un exemplu clasic de transmitere de matrice în funcții. Chiar dacă **gets()** din biblioteca dvs. standard este mai sofisticată şi mai complexă, următoarea versiune mai simplificată, numită **xgets()**, vă va da o idee despre modul ei de lucru.

```
/* O versiune foarte simpla a functiei
   gets() din biblioteca standard. */
char *xgets(char *s)
{
    char ch, *p;
    int t;

    p = s; /* gets() returneaza un pointer catre s */

    for(t=0; t<80; ++t) {
          ch = getchar();

          switch(ch) {
                case '\n':
                      s[t] = '\0'; /* incheie sirul */
                      return p;
                case '\b':
                      if(t>0) t--;
                      break;
                default:
                      s[t] = ch;
          }
    }
```

```
    s[80] = '\0';
    return p;
}
```

Funcţia **xgets()** trebuie apelată cu un pointer de tip caracter, care poate fi ori o variabilă declarată ca un pointer de tip caracter ori numele unei matrice de tip caracter, ce prin definiţie este un astfel de pointer. La intrare, **xgets()** generează o buclă **for** de la 0 la 80, ceea ce împiedică introducerea de la tastatură a şirurilor mai mari. Dacă sunt introduse mai mult de 80 de caractere, se iese din funcţie. (Funcţia iniţială **gets()** nu are această restricţie.) Deoarece C nu are construcţii proprii de control al limitelor, trebuie să vă asiguraţi că orice matrice foloseşte **gets()** poate accepta cel puţin 80 de caractere. Pe măsură ce scrieţi caracterele de la tastatură, ele sunt plasate în şir. Dacă apăsaţi Backspace, contorul **t** este decrementat cu 1, ştergând efectiv caracterul anterior din matrice. Când tastaţi **ENTER**, la sfârşitul şirului este plasat un caracter de null, semnalând încheierea sa. Deoarece matricea care apelează **xgets()** este modificată, la returnarea sa ea conţine caracterele pe care le tastaţi.

argc şi argv - Argumente pentru main()

Câteodată este util să transmiteţi informaţii într-un program când îl executaţi. În general, informaţiile se transmit funcţiei **main()** prin intermediul argumentelor din linia de comandă. *Un argument al liniei de comandă* este informaţia care urmează numele programului pe linia de comandă a sistemului de operare. De exemplu, când compilaţi programe, puteţi să scrieţi ceva de genul următor la prompterul ecranului:

cc *nume_program*

unde *nume_program* este un argument de linie de comandă care specifică numele programului pe care doriţi să îl compilaţi.

Există două argumente speciale proprii, **argc** şi **argv**, care sunt folosite pentru a primi argumentele liniei de comandă. Parametrul **argc** memorează numărul de argumente de pe linia de comandă şi este de tip întreg. Valoarea sa este cel puţin egală cu 1, deoarece numele programului este considerat drept primul argument. Parametrul **argv** este un pointer la o matrice de pointeri de tip caracter. Fiecare element al matricei indică spre un argument din linia de comandă. Toate argumentele liniei de comandă sunt şiruri - orice număr va fi convertit de program în formatul intern corespunzător. De exemplu, acest program simplu afişează pe ecran **Hello** şi numele dvs. dacă îl scrieţi direct după numele programului.

```
#include <stdio.h>
#include <stdlib.h>
```



```
void main(int argc, char *argv[])
{
    if(argc!=2) {
          printf("Ati uitat sa scrieti numele dvs. \");
    exit(1);
    }
    printf("Hello %s", argv[1]);
}
```

Dacă denumiţi acest program **nume** iar numele dvs. este Tom, pentru a rula programul trebuie să scrieţi **nume Tom**. Pe ecran va apărea **Hello Tom**.

În multe medii fiecare argument al liniei de comandă trebuie separat cu un spaţiu simplu sau cu unul de tabulare. Virgulele, punct şi virgulă şi cele asemenea lor nu sunt considerate separatori. De exemplu,

```
run Spot, run
```

este formată din trei şiruri, în timp ce

```
Ion,Vasile,George
```

este un singur şir, deoarece virgulele nu sunt, în general, acceptate ca separatori.

Unele medii vă permit să includeţi între ghilimele duble un şir care conţine spaţii, ceea ce face ca întregul şir să fie tratat ca un singur argument. Pentru detalii specifice privind definirea parametrilor liniei de comandă, consultaţi manualul sistemului dvs. de operare.

Trebuie să declaraţi corect **argv**. Cea mai uzuală metodă este:

```
char *argv[];
```

Parantezele drepte, între care nu este menţionat nimic, indică faptul că matricea are lungime nedeterminată. Acum puteţi avea acces la argumentele individuale, punând indici pentru **argv**. De exemplu, **argv[0]** indică spre primul şir, care este întotdeauna numele programului; **argv[1]** indică spre primul argument ş.a.m.d.

Un alt exemplu scurt care foloseşte argumentele liniei de comandă este programul numit **numarainvers**, prezentat în continuare. Porneşte numărarea de la o valoare dată (specificată pe linia de comandă) şi se îndreaptă în jos, spre 0, iar când ajunge aici emite un semnal sonor. Observaţi că primul argument conţinând numărul este convertit într-un întreg prin funcţia standard **atoi()**. Dacă al doilea argument al liniei de comandă este şirul "afişează", numărătoarea va fi afişată pe ecran.

```
/* Program de numarare inversa. */
#include <stdio.h>
#include <stdlib.h>
#include <ctype.h>
#include <string.h>

void main(int argc, char *argv[])
{
    int afis, contor;
    if(argc<2) {
          printf("Trebuie sa introduceti numarul
                   initial\n");
          printf("pe linia de comanda. Mai incercati o
                   data. \n");
          exit(1);
    }
    if(argc==3 && !strcmp(argv[2], "afiseaza")) afis = 1;
    else afis = 0;
    for(contor=atoi(argv[1]); contor; --contor)
          if(afis) printf("%d\n", contor);
    putchar('\a'); /* Aceasta va da un semnal sonor
                        pe majoritatea calculatoarelor */
    printf("Gata");
}
```

Rețineți că dacă nu a fost specificat nici un argument pe linia de comandă, este afişat un mesaj de eroare. Un program cu argumente pe linia de comandă oferă deseori mesaje dacă utilizatorul încearcă să ruleze programul fără să introducă informațiile corecte.

Pentru a avea acces la un caracter individual dintr-unul din şirurile de comandă, adăugați un al doilea indice lui **argv**. De exemplu, următorul program afişează toate argumentele cu care este apelată funcția, câte un caracter o dată.

```
#include <stdio.h>

void main(int argc, char *argv[])
{
    int t, i;

    for(t=0; t<argc; ++t) {
          i = 0;

          while(argv[t][i]) {
                putchar(argv[t][i]);
                ++i;
          }
    }
}
```


Rețineți că primul indice oferă accesul la şir, iar al doilea la caracterele individuale ale şirului.

De obicei, folosiți **argc** şi **argv** pentru a da programului dvs. comenzile inițiale. Teoretic, puteți avea până la 32.767 de argumente dar majoritatea sistemelor de operare nu permit decât câteva. În mod normal, utilizați aceste argumente pentru a indica un nume de fişier sau o opțiune. Utilizând argumentele liniei de comandă oferiți programului dvs. o înfățişare profesională şi facilitați folosirea sa în fişiere batch.

Când un program în C nu necesită parametri din linia de comandă, este o practică uzuală să declarați explicit **main()** ca neavând parametri, utilizând cuvântul-cheie **void** în lista sa de parametri. Aceasta este abordarea utilizată pentru multe dintre programele din **Partea Întâi** a acestei cărți. Totuşi, dacă doriți, puteți specifica simplu o listă vidă de parametri. (De altfel, în C++ este redundantă utilizarea lui **void** pentru a indica o listă de parametri vidă.)

Numele **argc** şi **argv** sunt folosite prin tradiție, dar numele lor este ales arbitrar. Puteți să numiți aceşti parametri ai lui **main()** oricum doriți. De asemenea, unele compilatoare permit în **main()** argumente în plus, aşa încât verificați manualul utilizatorului.

Instrucțiunea *return*

Instrucțiunea **return** are două utilizări importante. Prima determină ieşirea imediată din funcția în care se află. Aceasta înseamnă că determină reîntoarcerea execuției programului la codul care a apelat-o. A doua poate fi utilizată pentru a returna o valoare. Amândouă sunt studiate în acest paragraf.

Revenirea dintr-o funcție

Există două căi de încheiere a execuției unei funcții şi de revenire în programul apelant. Prima are loc atunci când este executată ultima instrucțiune a funcției şi se întâlneşte acolada de sfârşit (}). (Desigur, acolada nu este prezentă efectiv în codul obiect, dar puteți să o imaginați astfel.) De exemplu, funcția **afis_invers()** din acest program pur şi simplu afişează invers pe ecran şirul „Îmi place C" şi apoi revine.

```
#include <string.h>
#include<stdio.h>

void afis_invers(char *s);

void main(void)
{
    afis_invers("imi place C++");
}

void afis_invers(char *s)
{
    register int t;

    for(t=strlen(s)-1; t>0; t--) putchar(s[t]);
}
```

O dată afişat şirul, lui **afis_invers** nu i-a mai rămas nimic de făcut, astfel încât se întoarce în locul din care a fost apelată.

De fapt, nu sunt multe funcţiile care îşi încheie execuţia prin această metodă implicită. Majoritatea funcţiilor se bazează pe instrucţiunea **return** ca să-şi încheie execuţia ori deoarece trebuie returnată o valoare, ori pentru a face codul unei funcţii mai simplu şi mai eficient.

Reţineţi că o funcţie poate să conţină mai multe instrucţiuni **return**. De exemplu, funcţia **afla_subsir** din următorul program returnează poziţia de început a unui subşir dintr-un şir sau -1 dacă nu se găseşte subşirul.

```
#include <stdio.h>

int afla_subsir(char *s1, char *s2);

void main(void)
{
    if(afla_subsir("C este dragut", "este") != -1)
          printf("subsirul a fost gasit");
}
/* Returneaza indicele primei localizari a lui s2 in s1. */
afla_subsir(char *s1, char *s2)
{
    register int t;
    char *p, *p2;

    for(t=o; s1[t]; t++) {
          p = &s1[t];
          p2 = s2;

          while(*p2 && *p2==*p) {
                p++;
                p2++;
          }
          if(!*p2) return t; /* primaul return */
    }
    return -1; /* al doilea return */
}
```


Valori returnate

Toate funcţiile, cu excepţia celor de tip **void**, returnează o valoare. Această valoare este specificată explicit de către instrucţiunea **return**. Dacă nu există nici o instrucţiune **return**, atunci valoarea returnată de funcţie este teoretic nedefinită. (În general, implementarea compilatorului de C/C++ întoarce 0 când nu este specificată nici o valoare de returnat, dar nu trebuie să vă bazaţi pe acest lucru dacă doriţi să asiguraţi portabilitatea programului.) Cu alte cuvinte, atâta vreme cât o funcţie nu este declarată ca fiind **void**, puteţi să o folosiţi ca operand în orice expresie validă. Astfel, toate expresiile care urmează sunt corecte:

```
x = putere(y);
if(max(x,y) > 100) printf("maimare");
for(ch=getchar(); isdigit(ch); )...;
```

Însă, ca regulă generală, o funcţie nu poate fi ţinta unei atribuiri. O astfel de instrucţiune:

```
modific(x,y) = 100; /* instructiune incorecta */
```

este greşită. Compilatorul de C/C++ o va taxa ca eroare şi nu va compila un program care conţine aşa ceva. (După cum este prezentat în **Partea a doua**, C++ acceptă unele excepţii interesante de la această regulă, permiţând unor tipuri de funcţii să se găsească în membrul stâng al unei atribuiri.)

Când scrieţi programe, funcţiile dvs. vor fi, în general, de trei tipuri. Primul tip este de calcul simplu. Aceste funcţii sunt proiectate pentru a efectua operaţii asupra argumentelor lor şi a returna o valoare rezultată din aceste operaţii. O funcţie de calcul este o funcţie „pură". Exemple sunt funcţiile standard de bibliotecă **sqrt()** şi **sin()**, care calculează rădăcina pătrată şi respectiv sinusul argumentelor lor.

Al doilea tip de funcție manevrează informațiile și returnează o valoare care indică pur și simplu reușita sau nereușita acelei manevre. Un exemplu este funcția de bibliotecă **fclose()**, care este folosită pentru a închide un fișier. Dacă operația de închidere a decurs cu succes, funcția returnează 0; dacă operația nu a reușit, ea returnează un cod de eroare.

Ultimul tip de funcție nu are o valoare returnată explicit. În esență, funcția este strict de procedură și nu returnează o valoare. Un exemplu este **exit()**, care termină un program. Toate funcțiile care nu returnează valori trebuie declarate ca returnând tipul **void**. Declarând o funcție **void**, ea este ferită de a fi utilizată în expresii, evitându-se astfel manevrări accidentale greșite.

Uneori, funcții care nu produc un rezultat interesant returnează totuși ceva. De exemplu, **printf()** returnează numărul de caractere scrise. Totuși, ar fi neobișnuit ca să întâlniți un program care să folosească într-adevăr acest rezultat. Cu alte cuvinte, chiar dacă toate funcțiile, cu excepția celor **void**, returnează valori, nu trebuie neapărat să utilizați valorile returnate. O întrebare uzuală despre valorile returnate de funcții este: „Dacă se returnează o valoare nu trebuie să o atribui unei anumite variabile?" Răspunsul este nu. Dacă nu se specifică nici o atribuire, se renunță pur și simplu la rezultat. Să luăm următorul program, care folosește funcția **inmul()**.

```
#include <stdio.h>

int inmul(int a, int b);

void main(void)
{
    int x, y, z;

    x = 10; y = 20;
    z = inmul(x, y);                /* 1 */
    printf("%d", inmul(x, y));      /* 2 */
    inmul(x, y);
}

inmul(int a, int b)
{
    return a*b;
}
```

În linia 1, valoarea returnată de **inmul()** este atribuită lui **z**. În linia a doua, valoarea returnată nu este de fapt atribuită, dar este folosită de funcția **printf()**. În sfârșit, în linia 3, valoarea returnată se pierde deoarece nu este nici atribuită unei alte variabile nici utilizată ca parte a unei expresii.

Funcții care returnează valori ce nu sunt de tip întreg

Când tipul returnat de o funcție nu este declarat explicit, i se atribuie automat **int**. Acest lucru este valabil pentru multe funcții. Totuși, când este necesar alt tip de date, procesul implică doi pași. Primul, funcției trebuie să-i se ofere un specificator de tip explicit. Al doilea, tipul returnat de către funcție trebuie să fie specificat înainte de prima sa apelare. Doar în acest fel compilatorul generează codul corect pentru funcții care returnează valori ce nu sunt de tipul **int**.

Funcțiile pot fi declarate ca returnând orice tip valid de date (cu excepția matricelor). Declararea funcțiilor este similară declarării variabilelor: specificatorul de tip precede numele funcției. El spune compilatorului ce tip de date returnează acea funcție. Această informație este foarte importantă pentru rularea corectă a programului, deoarece tipuri de date diferite au diferite mărimi și reprezentări interne.

Tipul unei funcții care nu returnează întregi trebuie să fie cunoscut de restul programului înainte de a folosi acea funcție. Aceasta deoarece, dacă nu i se spune altfel, compilatorul va presupune că funcția returnează o valoare întreagă. Dacă programul apelează o funcție care returnează alt tip și ea încă nu a fost declarată, compilatorul generează un cod greșit pentru apelarea funcției. Pentru a preveni acest lucru, trebuie să folosiți o formă specială a instrucțiunii de declarare, aproape de începutul programului, pentru a spune compilatorului tipul efectiv al valorii returnate de acea funcție.

Există două căi de declarare a unei funcții înainte de a fi folosită: calea clasică și metoda modernă a prototipului. Calea clasică era singura metodă permisă atunci când a fost inventat C, dar acum este depășită. Prototipurile au fost adăugate de standardul ANSI C. Calea clasică este încă permisă de standardul ANSI C pentru a oferi compatibilitate cu codul vechi, dar utilizarea sa în continuare nu este recomandată.



***NOTĂ:** C++ nu permite calea clasică de declarare a funcției, ci cere în locul ei prototipuri. Materialul prezentat în acest paragraf se aplică doar limbajului C.*

Acest paragraf descrie pe scurt metoda clasică de declarare a funcțiilor, deși ea este depășită. Chiar așa fiind, multe programe existente încă o mai folosesc, deci trebuie să vă familiarizați cu ea. Mai mult, metoda prototipului este în mare o extensie a conceptului tradițional. (Prototipurile funcțiilor sunt discutate în paragraful următor.)

Utilizând metoda demodată de declarare a funcțiilor, specificați tipul valorii și numele funcției returnate la începutul programului pentru a informa compilatorul că o funcție va returna un anumit tip de valoare, alta decât una întreagă, așa cum se arată aici:

```
#include <stdio.h>

float sum(); /* declara functia */
float prima, adoua;

void main(void)
{
    prima = 123.23;
    adoua = 99.09;

    printf("%F", sum());
}

float sum()
{
    return prima + adoua;
}
```

Prima declarație de tip de funcție spune compilatorului că **sum()** returnează un tip de date în virgulă mobilă. Aceasta permite compilatorului să genereze codul corect al apelărilor lui **sum()**. Fără declarare, compilatorul va afişa o eroare de nepotrivire.

Instrucțiunea clasică de declarare a tipului funcției are forma generală:

specificator_de_tip nume_funcție();

Rețineți că lista de parametri este vidă. Chiar dacă funcția urmează să preia argumente, nici unul nu va fi prezent în declararea tipului.

Fără instrucțiunea de declarare a tipului va apărea o nepotrivire între tipul de date returnat de funcție şi tipul de date pe care îl aşteaptă rutina. Rezultatele vor fi bizare şi neprevăzute. Dacă ambele funcții sunt în acelaşi fişier, compilatorul găseşte nepotrivirea şi nu compilează programul. Dar, dacă funcțiile sunt în fişiere diferite, compilatorul nu depistează eroarea. În C verificarea tipului nu se face în timpul editării legăturilor sau al rulării, ci doar în timpul compilării. Din acest motiv, trebuie să fiți siguri că tipurile sunt compatibile.

*NOTĂ: Când este returnat un caracter dintr-o funcție declarată ca fiind de tip **int**, valoarea acestuia este convertită în întreg. Deoarece C lucrează curat cu conversia caracterelor în întregi şi invers, o funcție care returnează o valoare de tip caracter este declarată rareori ca returnând o valoare de acest tip. Programatorul se bazează pe tipul de conversie implicită a caracterelor în întregi şi invers. Acest lucru se întâlneşte frecvent în vechile coduri în C şi teoretic nu este considerat o greşeală.*

Prototipurile funcțiilor

Declararea clasică a funcțiilor (descrisă în paragraful precedent) permite doar să fie declarat tipul returnat de o funcție. Standardul ANSI C extinde declararea clasică a funcției permițând să fie declarate în afară de tipul returnat şi numărul şi tipul argumentelor funcției. Această definiție extinsă este denumită *prototipul funcției*. Cum s-a mai spus, prototipurile de funcții nu făceau parte din limbajul C original. Ele sunt una dintre cele mai importante îmbunătățiri ale limbajului C făcute prin standardizare. De asemenea, în C++ ele sunt *impuse*. Toate exemplele din această carte includ prototipuri complete de funcții. Prototipurile permit limbajului C să asigure verificare mai strictă de tipuri, asemănătoare celor realizate de limbaje precum Pascal. Când folosiți prototipurile, compilatorul poate să găsească şi să afişeze orice conversie ilegală între tipurile argumentelor folosite pentru a apela o funcție şi definiția tipurilor parametrilor săi. De asemenea, compilatorul va depista diferențele dintre numărul de argumente folosit pentru apelare şi numărul parametrilor acelei funcții.

Forma generală a definirii unui prototip este:

tip nume_funcție(tip nume_param1, tip nume_param2,..., tip nume_paramN);

Folosirea numelor parametrilor este opțională. Dar, ele permit compilatorului să identifice prin nume orice nepotrivire de tipuri atunci când apare o eroare, astfel încât este o idee bună să le includem.

Următorul program ilustrează importanța prototipurilor funcțiilor. El emite un mesaj de eroare deoarece se încearcă o apelare a funcției **la_patrat()** cu un argument întreg în loc de pointerul de tip întreg necesar. (Nu este permisă convertirea unui întreg într-un pointer.)

```
/* Acest program utilizeaza prototipul functiei
   pentru a forta o verificare stricta a tipului */
void la_patrat(int *i); /* prototip */

void main(void)
{
    int x;

    x = 10;
    la_patrat(x); /* nepotrivire de tip */
}

void la_patrat(int *i)
{
    *i = *i * *i;
}
```

Datorită necesităţii compatibilităţii cu versiunea originală pentru C, prototipurilor funcţiilor li se aplică unele reguli speciale. Prima, când este declarat tipul returnat de funcţie, dar lista de parametri este vidă, compilatorul presupune că nu se dă nici o *informaţie referitoare la parametri*. În ceea ce priveşe compilatorul, funcţia poate avea mai mulţi parametri sau nici unul. Dar ce face un prototip de funcţie care nu are nici un parametru? Iată răspunsul: când o funcţie nu are parametri, prototipul ei foloseşte **void** în interiorul listei de parametri. De exemplu, dacă o funcţie numită **f()** returnează un **float** şi nu are parametri, prototipul ei arată astfel:

```
float f(void);
```

Aceasta spune compilatorului că funcţia nu are parametri şi că orice apelare a ei care conţine parametri este o eroare.



***NOTĂ:** În C++ **f()** şi **f(void)** sunt echivalente.*

Introducerea prototipurilor afectează promovarea de tip automată în C. Când este apelată o funcţie fără prototip, toate caracterele sunt convertite în întregi şi toate variabilele **float** în **double**. Aceste promovări cam ciudate de tip sunt legate de anumite caracteristici ale mediului original în care s-a dezvoltat C. Totuşi, dacă daţi prototipul unei funcţii, tipul specificat în el este menţinut şi nu mai are loc nici o promovare a tipului.

Prototipurile funcţiilor vă ajută să ocoliţi greşelile înainte ca ele să apară. În plus, vă dau posibilitatea să verificaţi dacă programul lucrează corect, nepermiţând funcţiilor să fie apelate cu argumente nepotrivite.

Ţineţi foarte bine minte: chiar dacă utilizarea prototipurilor de funcţii este recomandată cu tărie în C, lipsa acestora nu este o greşeală. Această toleranţă este necesară pentru a admite codul C anterior apariţiei prototipului. Chiar şi aşa, codul dvs. trebuie, în general, să includă informaţii de prototipuri complet definite. Discuţia anterioară despre metoda clasică de declarare a funcţiilor este inclusă în această carte doar de dragul de a fi exhaustivi.



***REŢINEŢI:** Chiar dacă prototipurile sunt opţionale în C, ele sunt impuse de C++. Aceasta înseamnă că orice funcţie într-un program în C++ trebuie să fie declarată ca prototip.*

Returnarea pointerilor

Chiar dacă funcţiile care returnează pointeri sunt manevrate la fel ca oricare tip de funcţie, este necesar să discutăm câteva aspecte importante.

Pointerii la variabile nu sunt nici întregi, nici întregi fără semn. Ei sunt adresele din memorie ale unui anumit tip de date. Motivul acestei distincţii este acela că aritmetica pointerilor se bazează pe tipul respectiv. De exemplu, dacă este incrementat un pointer de tip întreg, el va conţine o valoare mai mare cu 2 decât cea iniţială (presupunând un întreg de 2 octeţi). În general, de fiecare dată când este incrementat (sau decrementat) un pointer, el va indica spre următorul (sau anteriorul) element de date de tipul său. Deoarece fiecare tip de date poate fi de o lungime diferită, compilatorul trebuie să ştie către ce tip de date indică pointerul. Din acest motiv, o funcţie care returnează un pointer trebuie să declare explicit tipul acestuia.

Pentru a returna un pointer, o funcţie trebuie să fie declarată ca atare. De exemplu, această funcţie returnează un pointer spre prima apariţie a caracterului **c** în şirul **s**:

```
/* Returneaza un pointer la prima aparitie a lui c in s. */
char *gaseste(char c, char *s)
{
   while(c!=*s && *s) s++;
   return(s);
}
```

Dacă nu se găseşte nimic, este returnat un pointer către caracterul de null de sfârşit. Iată un scurt program care foloseşte **gaseste()**:

```
#include <stdio.h>

char *gaseste(char c, char *s); /* prototip */

void main(void)
{
   char s[80], *p, ch;

   gets(s);
   ch = getchar();
   p = gaseste(ch, s);

   if(*p) /* potrivire */
         printf("%s", p);
   else
         printf("Nu s-a gasit nici o potrivire.");
}
```

Programul citeşte un şir şi apoi un caracter. Dacă acel caracter există în şir, programul afişează şirul din punctul în care l-a găsit. Altfel, el afişează **Nu s-a gasit nici o potrivire**.

Funcţii de tipul void

Una dintre utilizările tipului **void** este să declare explicit funcţii care nu returnează valori. Aceasta previne folosirea lor într-o expresie şi ajută la avertizarea asupra unor utilizări accidentale greşite. De exemplu, funcţia **afis_vertical()** afişează şirul care îi serveşte drept argument vertical, în partea de jos a ecranului. Deoarece nu returnează nici o valoare, ea este declarată ca fiind de tip **void**.

```
void afis_vertical(char *sir)
{
    while(*sir)
          printf("%c\n", *sir++);
}
```

Înainte de a folosi orice funcţie de tip **void**, trebuie să îi declaraţi prototipul. Dacă nu o faceţi, compilatorul presupune că ea returnează un întreg iar când va ajunge la funcţia efectivă va declara o nepotrivire de tipuri. Următorul program arată un exemplu corect care afişează vertical pe ecran un singur argument al liniei de comandă.

```
#include <stdio.h>

void afis_vertical(char *sir); /* prototip */

void main(int arcg, char *argv[])
{
    if(argc) afis_vertical(argv[1]);
}

void afis_vertical(char *sir)
{
    while(*sir)
          printf("%c\n", *sir++);
}
```

Înainte ca standardul ANSI C să definească **void**, funcţiile care nu returnau valori erau considerate implicit ca având tip de returnare **int**. De aceea, să nu fiţi surprinşi să vedeţi multe astfel de exemple în vechile coduri.

Ce returnează main() ?

Funcţia **main()** returnează un întreg către sistemul care a apelat-o, care este în general sistemul de operare. Returnarea unei valori din **main()** este echivalentă cu apelarea lui **exit()** cu aceeaşi valoare. Dacă **main()** nu returnează explicit o valoare, valoarea transmisă sistemului care a apelat-o este teoretic nedefinită. În practică, majoritatea compilatoarelor de C/C++ returnează automat 0, dar nu vă bazaţi pe aceasta dacă urmăriţi portabilitatea programului.

De asemenea, dacă funcţia **main()** nu returnează o valoare puteţi să o declaraţi ca fiind de tip **void**. (Multe programe din această carte folosesc acest lucru.) Unele compilatoare emit un mesaj de avertizare dacă o funcţie nu este declarată de tip **void** deşi nu returnează o valoare. Astfel, dacă decideţi că **main()** nu returnează nici o valoare, va trebui să declaraţi tipul său rezultat ca fiind **void**.

Recursivitate

În C o funcţie poate să se apeleze singură. Se spune că o funcţie este *recursivă* dacă o instrucţiune din corpul ei apelează chiar acea funcţie. Recursivitatea este procesul de definire a unui lucru prin el însuşi şi, câteodată, este numită *definiţie circulară*.

Funcţia **fact()** este un exemplu simplu de funcţie recursivă care calculează factorialul unui întreg. Factorialul unui număr **n** este produsul tuturor numerelor între 1 şi **n**. De exemplu, 3 factorial este 1x2x3, deci 6. În continuare, sunt arătate atât **fact()** cât şi echivalentul sau iterativ:

```
fact(int n) /* recursiva */
{
    int raspuns;

    if(n==1) return(1);
    raspuns = fact(n-1)*n; /*apelare recursiva */
    return(raspuns);
}

fact(int n)  /*ne-recursiva */
{
    int t, raspuns;

    raspuns = 1;
    for(t=1; t<=n; t++)
    raspuns = raspuns*t;
    return(raspuns);
}
```

Ar trebui ca versiunea nerecursivă a funcţiei **fact()** să fie clară. Ea foloseşte o buclă care rulează de la 1 la **n** şi înmulţeşte progresiv fiecare număr cu produsul rezultat anterior.

Operarea cu funcţia recursivă **fact()** este puţin mai complexă. Când **fact()** este apelat cu argumentul 1, funcţia returnează 1. Altfel, ea returnează produsul **fact(n-1)*n**. Pentru a evalua această expresie, **fact()** este apelat cu **n-1**. Aceasta se întâmplă până când **n** este egal cu 1 iar apelările funcţiei încep să fie returnate.

De exemplu, când se calculează factorialul lui 2, prima apelare a lui **fact()** determină o a doua, apelare recursivă cu argumentul 1. Această apelare returnează 1, care este apoi înmulţit cu 2 (valoarea iniţială a lui **n**). Răspunsul va fi 2. Încercaţi să efectuaţi calculul complet pentru 3 factorial de unul singur. (Poate doriţi să introduceţi instrucţiuni **printf()** în **fact()** pentru a vedea nivelul fiecărei apelări şi care sunt răspunsurile intermediare.)

Când o funcţie se apelează pe ea însăşi, în memoria stivă sunt alocate zone pentru un nou set de variabile locale şi parametri, iar codul funcţiei se execută cu aceste noi variabile din vârf. O apelare recursivă nu creează o nouă copie a funcţiei. Doar argumentele sunt noi. Pe măsură ce apelările recursive sunt returnate, variabilele locale vechi şi parametrii sunt îndepărtaţi din memoria stivă iar execuţia se întoarce la punctul în care funcţia s-a apelat singură. Funcţiile recursive sunt denumite „telescop" (deoarece se „întind" şi se „strâng").

Majoritatea rutinelor recursive nu reduc semnificativ mărimea codului, nici nu îmbunătăţesc utilizarea memoriei. Dimpotrivă, versiunile recursive pentru majoritatea rutinelor pot fi executate puţin mai încet decât echivalentul lor iterativ datorită apelărilor suplimentare ale funcţiei. De fapt, multe apelări recursive ale unei funcţii pot determina depăşirea limitei memoriei stivă. Deoarece memorarea parametrilor funcţiei şi a variabilelor locale se face în memoria stivă şi fiecare nouă apelare creează o nouă copie a acestor variabile, acestea s-ar putea scrie în stivă peste alte date sau peste program. Totuşi, probabil că nu este cazul să vă fie teamă decât dacă funcţia recursivă nu rulează exagerat.

Avantajul principal al funcţiilor recursive este acela că puteţi să le folosiţi pentru a crea versiuni mai simple şi mai clare ale unor algoritmi. De exemplu, algoritmul scurt de sortare este destul de dificil de introdus în mod iterativ. De asemenea, unele probleme, în special cele care se adresează inteligenţei artificiale, se pretează soluţiilor recursive. În sfârşit, unele persoane par să gândească mai degrabă recursiv decât iterativ.

Când scrieţi funcţii recursive trebuie să aveţi undeva o instrucţiune **if** pentru a forţa funcţia să se reîntoarcă fără ca apelarea recursivă să mai aibă loc. Dacă nu faceţi aceasta, o dată apelată, funcţia nu se va mai încheia. Omiterea lui **if** este o greşeală uzuală la scrierea unei funcţii recursive. Utilizaţi **printf()** şi **getchar()** liber în timpul derulării programului astfel încât să puteţi urmări ce se întâmplă şi să renunţaţi la execuţie dacă vedeţi vreo greşeală.

Declararea listelor cu număr variabil de parametri

Puteţi să specificaţi funcţii care au un număr variabil de parametri. Cel mai simplu exemplu este **printf()**. Pentru a spune compilatorului că funcţia va primi un număr necunoscut de parametri, trebuie să încheiaţi declararea parametrilor săi cu trei puncte. De exemplu, acest prototip specifică faptul că **func()** va avea cel puţin doi parametri de tip întreg şi un număr necunoscut (inclusiv 0) de parametri după acesta.

```
func(int a, int b, ...);
```

Această formă de declarare este utilizată, de asemenea, la definirea unei funcţii.

Orice funcţie care foloseşte un număr variabil de parametri trebuie să aibă cel puţin un parametru. De exemplu, aceasta este o formă incorectă:

```
func(...); /* gresit */
```

Declaraţii de parametri pentru funcţii în mod clasic şi modern

Versiunea originală a limbajului C folosea o metodă diferită de declaraţii de parametri, uneori numită forma *clasică*. Această carte foloseşte un tip de declaraţie numită forma *modernă*. Standardul ANSI C admite ambele forme, dar recomandă cu tărie pe cea modernă. Standardul propus pentru ANSI C++ nu admite decât metoda modernă pentru declararea parametrilor. Totuşi, ar trebui să o cunoaşteţi şi pe cea clasică deoarece multe din programele vechi în C încă o mai folosesc. (Dar nu este cazul să o utilizaţi în codurile pe care le scrieţi.)

Declararea clasică a parametrilor funcţiei constă în două părţi: o listă de parametri, care se găseşte între parantezele care urmează numele funcţiei şi declararea efectivă a parametrilor care se află între paranteza închisă şi acolada deschisă pentru începutul funcţiei. Aici este prezentată forma generală a definirii clasice de parametri:

tip nume_funcţie(param1, param2, ...paramN)
tip param1;
tip param2;

.
tip paramN;
{
cod funcţie
}

De exemplu, această declarare modernă:

```
float f(int a, int b, char ch)
{
    /*... */
}
```

va arăta astfel în formă clasică:

```
float f(a, b, ch)
int a, b;
char ch;
{
    /* ... */
}
```

Reţineţi că forma clasică permite declararea a mai mult de un parametru în listă după numele tipului.

REŢINEŢI: Forma clasică a declarării de parametri este demodată în limbajul C şi neadmisă de C++.

Caracteristici de implementare

Atunci când creaţi funcţii, există câteva lucruri importante care afectează eficienţa şi utilizarea lor şi pe care trebuie să vi le amintiţi. Aceste probleme sunt subiectul prezentului paragraf.

Parametri şi funcţii de utilitate generală

O funcţie de utilitate generală este una care va fi utilizată într-o varietate de situaţii, probabil în mai multe programe. În mod normal nu trebuie să vă bazaţi funcţiile de interes general pe date globale. Dacă este posibil, ar trebui ca toate informaţiile pe care le necesită o funcţie să fie introduse prin parametrii săi.
În afară de a permite ca funcţiile dvs. să fie de interes general, parametrii fac codul lizibil şi mai puţin vulnerabil la erori rezultate prin efecte secundare.



Eficienţă

Funcţiile sunt elementele de construcţie ale lui C şi sunt esenţiale pentru toate programele, mai puţin cele simple. Totuşi, în anumite aplicaţii specializate, s-ar putea să fie nevoie să eliminaţi o funcţie şi să o înlocuiţi cu un cod *inline*. Codul inline efectuează aceleaşi acţiuni ca şi o funcţie dar fără dezavantajul produs de o apelare de funcţie. Din acest motiv, dacă viteza de execuţie este foarte importantă, codul inline este deseori utilizat în locul unei apelări de funcţii.
Codul inline este mai rapid decât o apelare de funcţie din două motive. Primul, o instrucţiune de apelare ia un timp pentru a fi executată. Al doilea, dacă există argumente de transmis ele trebuie plasate în memoria stivă, ceea ce din nou consumă timp. Pentru majoritatea aplicaţiilor această mică îmbunătăţire a timpului de execuţie nu este semnificativă. Dar dacă totuşi este, reţineţi că fiecare apelare a funcţiei ia un timp pe care l-aţi putea salva plasând inline codul funcţiei. De exemplu, iată două versiuni ale aceluiaşi program care afişează pătratul numerelor de la 1 la 10. Versiunea inline rulează mai repede decât cealaltă deoarece apelarea funcţiei consumă timp.

Inline

```
#include <stdio.h>


void main(void)
{
    int x;

    for(x=1; x<11; ++x)

            printf("%d", x*x);
    printf("%d", patrat(x));
}
```

Apelare de funcţie

```
#include <sydio.h>
int patrat(int a);

void main(void)
{
        int x;

        for(x=1; x<11; ++x)
        printf("%d", patrat(x));


}

patrat(int a)
{
        return a*a;
}
```

NOTĂ: În C++ conceptul de funcţie inline este bine dezvoltat şi reglat. De fapt, funcţiile inline sunt o componentă importantă a limbajului C++.

Capitolul 7

Structuri, uniuni, enumerări şi tipuri definite de utilizator



Limbajul C vă permite să creați tipuri de date uzuale în cinci moduri diferite. Primul este *structura*, care grupează mai multe variabile sub același nume și este numită un tip de date *compozit*. (Mai sunt folosiți și termenii *agregat* și *conglomerat*.) Al doilea tip definit de utilizator este *câmpul de biți*, care este o variațiune a structurii ce permite acces ușor la biții individuali. Al treilea este *uniunea*, care face posibil ca aceleași zone de memorie să fie definite ca două sau mai multe tipuri diferite de variabile. Al patrulea tip de date uzuale este *enumerarea*, care este o listă de constante întregi cu nume. Ultimul tip definit de utilizator este creat prin utilizarea instrucțiunii **typedef** și definește un nou nume pentru un tip existent.

Structuri

O structură este un grup de variabile unite sub același nume, ce pune la dispoziție un mod convenabil de păstrare a informațiilor legate între ele. O *declarare de structură* formează un șablon care poate fi folosit pentru a crea structuri. Variabilele care fac parte din structură sunt denumite *membri* ai structurii. (Membrii structurii mai sunt numiți uzual și *elemente* sau *câmpuri*.)

În general, toți membrii structurii au o legătură logică. De exemplu, informațiile despre nume și adresă dintr-o listă pentru corespondență sunt reprezentate în mod normal într-o structură. Următorul fragment de cod prezintă cum se declară o structură care definește câmpurile pentru nume și adresă. Cuvântul-cheie **struct** spune compilatorului că a fost declarată o structură.

```
struct adrese
{
char nume[30];
char strada[40];
char oras[20];
char judet[3];
unsigned long int cod;
};
```

Rețineți că declarația se termină cu punct și virgulă, deoarece este o instrucțiune. Numele **adrese** identifică această structură de date particulară și specificatorul său de tip.

Până aici *nu a fost creată efectiv nici o variabilă*. A fost definită doar forma datelor. Pentru a declara o variabilă de tipul **adrese** scrieți:

```
struct adrese adr_inform;
```


Aceasta declară o variabilă de tip **struct adrese** numită **adr_inform**. Când definiți o structură, definiți nu o variabilă, ci un tip compus de variabile. Nu va exista nici o variabilă până când nu veți declara una de acel tip.



***NOTĂ:** În C++, o dată ce a fost declarată o structură, puteți să declarați variabile de acel tip folosind doar numele ei generic, fără să o precedați cu cuvântul-cheie **struct**. De exemplu, pentru a declara în C++ o variabilă structură de tip **adrese**, veți scrie;*

```
adrese adr_inform;
```

Motivul acestei diferențe între C și C++ este acela că în C un nume generic de structură nu este un nume de tip complet, în timp ce în C++ este. Rețineți, totuși, că într-un program în C++ declarația de tip C este perfect legală.

Când este declarată o variabilă de tip structură (așa cum este **adr_inform**), compilatorul de C/C++ alocă automat suficientă memorie pentru a face față tuturor membrilor săi. **Figura 7-1** prezintă cum apare în memorie **adr_inform** presupunând că lungimea caracterelor este de un octet iar cea a întregilor lungi, de 4 octeți.

Când declarați o structură puteți declara una sau mai multe variabile de acest tip. De exemplu,

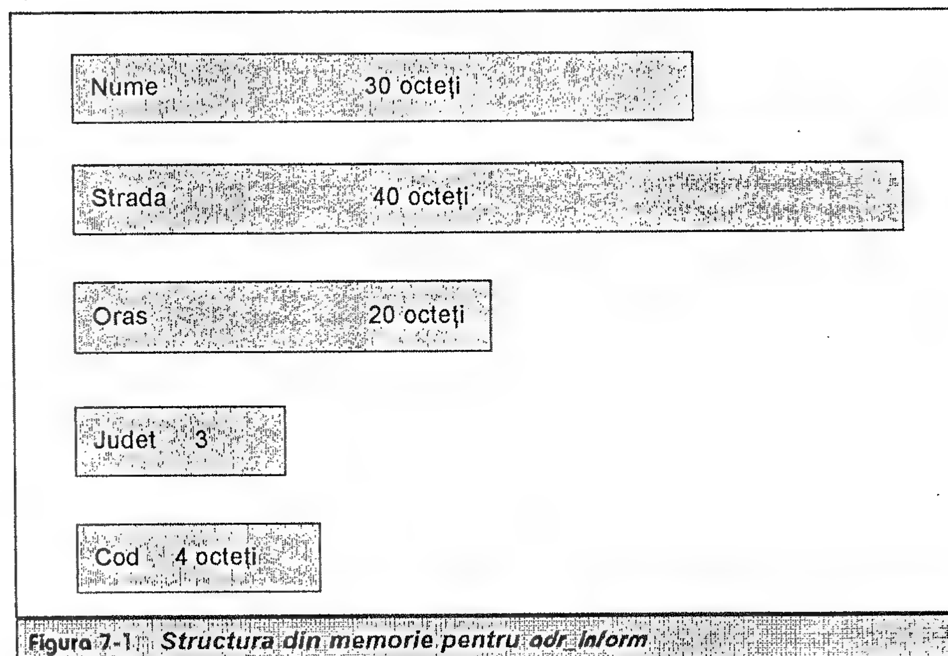


Figura 7-1 ***Structura din memorie pentru adr_inform***

```
struct adrese {
char nume[30];
char strada[40];
char oras[20];
char judet[3];
unsigned long int cod;
}   adr_inform, binform, cinform;
```

defineşte un tip de structură numită **adrese** şi declară ca fiind de acest tip variabilele **adr_inform, binform** şi **cinform**.

Dacă doriţi doar o variabilă numită **adr_inform**, nu mai este necesar numele generic. Aceasta înseamnă că:

```
struct {
    char nume[30];
    char strada[40];
    char oras[20];
    char judet[3];
    unsigned long int cod;
}   adr_info;
```

declară o variabilă numită **adr_inform** definită de structura care o precede.

Forma generală a declarării unei structuri este:

struct *nume_generic {*
tip nume_membru;
tip nume_membru;
tip nume_membru;
.
.
.
variabile_structura;

unde atât *nume_generic* cât şi *nume_variabila* pot fi omise, dar nu ambele simultan.

Accesul la membrii structurii

Accesul la membrii individuali ai unei structuri se face prin folosirea operatorului **.** (denumit uzual *operatorul punct*). De exemplu, următoarea linie atribuie valoarea 12345 câmpului **cod** al variabilei de tip structură **adr_inform** declarate mai devreme:


```
adr_inform.cod = 12345;
```

Numele variabilei tip structură urmat de un punct şi numele membrului se referă la acel membru individual. Forma generală de acces la un membru al structurii este:

nume_structură.nume_membru

De aceea, pentru a afişa **cod**-ul pe ecran, scrieţi:

```
printf("%d", adr_inform.cod);
```

Aceasta va afişa conţinutul din membrul **cod** al variabilei de tip structură **adr_inform**.

În acelaşi fel, poate fi folosită matricea de caractere **adr_inform.nume** pentru o apelare a lui **gets()**, aşa cum se arată aici:

```
gets(adr_inform.nume);
```

Aceasta transmite un pointer de tip caracter la începutul lui **nume**.

Deoarece **nume** este o matrice de caractere, puteţi avea acces la caracterele individuale ale lui **adr_inform.nume** punând indici la **nume**. De exemplu, puteţi să afişaţi conţinutul lui **adr_inform.nume** un caracter o dată, folosind următorul cod:

```
register int t;

for(t=0; adr_inform.nume[t]; ++t)
    putchar(adr_inform.nume[t]);
```

Atribuiri în structuri

Informaţia conţinută într-o structură poate fi atribuită unei alte structuri de acelaşi tip folosind o singură instrucţiune de atribuire. Aceasta înseamnă că nu trebuie să atribuiţi valoarea fiecărui membru separat. Următorul program ilustrează atribuirea în structuri.

```
#include <stdio.h>

void main(void)
{
   struct {
```

```
        int a;
        int b;
        } x, y;

    x.a = 10;
    y = x; /* atribuie o structura alteia */
    printf("%d", y.a);
}
```

După atribuire, **y.a** va conține valoarea 10.

Matrice de structuri

Probabil că cea mai uzuală întrebuințare a structurilor este cea din matricele de structuri. Pentru a declara o matrice de structuri, trebuie mai întâi să definiți o structură și apoi să declarați o variabilă de tip matrice de acel tip. De exemplu, pentru a declara o matrice de structuri cu 100 de elemente de tip **adrese**, definite mai devreme, scrieți:

```
struct adrese adr_inform[100];
```

Aceasta creează 100 de seturi de variabile care sunt organizate așa cum au fost definite în structura **adrese**.

Pentru a avea acces la o anumită structură se pun indici la numele matricei. De exemplu, pentru a afișa **cod**-ul din structura a treia, scrieți:

```
printf("%d", adr_inform[2].cod);
```

Ca toate variabilele de tip matrice, cele de structuri încep indicii de la 0.

Transmiterea structurilor către funcții

Acest paragraf prezintă transmiterea structurilor și a membrilor lor către funcții.

Transmiterea membrilor structurilor către funcții

Când transmiteți un membru al unei structuri unei funcții, de fapt transferați funcției valoarea acelui membru. De aceea, transmiteți o singură variabilă (dacă, nu cumva, desigur, acel element nu este compus, așa cum este o matrice de caractere). Să luăm, de exemplu, structura:


```
struct fane
{
    char x;
    int y;
    float z;
    char s[10];
}   mihai;
```

Iată exemple de transmitere a fiecărui membru către o funcție:

```
func(mihai.x);        /* paseaza valoarea de tip caracter
                         din x */
func2(mihai.y);       /* paseaza valoarea intreaga din y */
func3(mihai.z);       /* paseaza valoarea de tip float din z */
func4(mihai.s);       /* paseaza adresa sirului s */
func(mihai.s[2]);     /* paseaza valoarea de tip caracter
                         din s[2] */
```

Dacă doriți să transmiteți *adresa* unui membru individual al structurii, puneți operatorul **&** înaintea numelui structurii. De exemplu, pentru a transmite adresele membrilor structurii **mihai**, scrieți:

```
func(&mihai.x);       /* paseaza adresa caracterului x */
func2(&mihai.y);      /* paseaza adresa intregului y */
func3(&mihai.z);      /* paseaza adresa lui float z */
func4(mihai.s);       /* paseaza adresa sirului s */
func(&mihai.s[2]);    /* paseaza adresa caracterului s[2] */
```

Rețineți că operatorul **&** precede numele structurii, nu pe cel al membrului individual. Mai rețineți și că **s** deja este o adresă, deci nu mai necesită **&**.

Transmiterea structurilor întregi către funcții

Când o structură este folosită ca argument al unei funcții, întreaga structură este transmisă folosind metoda standard de apelare prin valoare. Desigur, aceasta înseamnă că orice modificare a conținutului structurii din interiorul funcției căreia îi este pasat nu afectează structura utilizată ca argument.

Când folosiți o structură ca parametru, amintiți-vă că tipul argumentului trebuie să corespundă tipului parametrului. De exemplu, în următorul program atât argumentul **arg** cât și parametrul **param** sunt declarate ca fiind de același tip de structură.

```
#include <stdio.h>

/* defineste un tip de structura. */
struct tip_struct {
    int a, b;
    char ch;
} ;

void f1(struct tip_struct param);

void main(void)
{
    struct tip_struct arg;
    arg.a = 1000;
    f1(arg);
}

void f1(struct tip_struct param)
{
    printf("%d", param.a);
}
```

După cum ilustrează acest program, dacă veți declara parametrii ca fiind structuri, va trebui să faceți declarații de structuri de tip global, astfel încât toate secțiunile programului să le poată folosi. De exemplu, dacă ați fi declarat **tip_struct** în interiorul lui **main()**, atunci el nu ar fi fost vizibil pentru **f1**.

Așa cum tocmai am afirmat, când transmiteți structuri, tipul argumentului trebuie să se potrivească cu tipul parametrului. Nu este suficient ca ei să fie efectiv similari; numele tipului lor trebuie să corespundă. De exemplu, următoarea versiune a programului precedent este incorectă și nu va fi compilată deoarece numele tipului argumentului folosit pentru apelarea lui **f1()** diferă de numele tipului parametrului său.

```
/* Acest program este incorect si nu va fi compilat */
#include <stdio.h>

/* Defineste un tip de structura. */
struct tip_struct {
    int a, b;
    char ch;
};

/* Defineste o structura similara cu tip_struct
   dar cu un nume diferit. */
struct tip2_struct {
    int a, b;
    char ch;
};

void f1(struct tip2_struct param);

void main(void)
{
    struct tip_struct arg;
    arg.a = 1000;
    f1(arg);    /* nepotrivire de tipuri */
}

void f1(struct tip_struct param)
{
    printf("%d", param.a);
}
```


Pointeri la structuri

C permite pointeri la structuri la fel cum permite pentru orice alt tip de variabilă. Totuși, există câteva aspecte speciale ale pointerilor la structuri pe care ar trebui să le cunoașteți.

Declararea unui pointer la o structură

Ca și alți pointeri, pointerii pentru structuri sunt declarați prin plasarea lui * în fața numelui variabilei de tip structură. De exemplu, considerând structura definită mai devreme **adrese**, următoarea instrucțiune declară **adr_pointer** ca un pointer către datele de acel tip.

```
struct adrese *adr_pointer;
```

Amintiți-vă că în C++ nu este necesar să plasați cuvântul-cheie **struct** la începutul declarației.

Utilizarea pointerilor la structuri

Există două întrebuințări principale ale pointerilor pentru structuri: generarea unei apelări de funcție prin parametri de referință și crearea listelor înlănțuite și a altor structuri de date dinamice folosind sistemul de alocare dinamică din C. Acest

capitol acoperă prima utilizare.

Există o piedică majoră în calea transmiterii structurilor, în afară de cele simple, către funcţii: suprasolicitarea datorită necesităţii de a plasa structura în memoria stivă atunci când este executată o apelare a funcţiei. (Amintiţi-vă că argumentele sunt transmise funcţiilor prin acest tip de memorie.) Pentru structuri simple, cu puţini membri, acest lucru nu este greu. Dacă, însă, structura conţine mulţi membri, sau dacă unii din ei sunt matrice, performanţele privind timpul de rulare pot să scadă la nivele inacceptabile. Soluţia pentru această problemă este să pasaţi funcţiei doar un pointer.

Când un pointer al unei structuri este pasat unei funcţii, în memoria stivă se reţine doar adresa structurii. Acest lucru garantează apelări de funcţii foarte rapide. În unele cazuri există şi al doilea avantaj, atunci când o funcţie trebuie să se adreseze structurii efective utilizate ca argument şi nu unei copii a ei. Prin transmiterea unui pointer, funcţia poate să modifice conţinutul structurii folosite la apelare.

Pentru a găsi adresa unei variabile de tip structură, plasaţi operatorul **&** înainte de numele acesteia. De exemplu, dându-se următorul fragment,

```
struct bal {
    float bilant;
    char nume[80];
}   persoana;
struct bal *p; /* declara un pointer la structura */
```

atunci

```
p = &persoana;
```

plasează adresa structurii **persoana** în pointerul **p**.

Pentru a avea acces la membrii structurii folosind un pointer la aceasta, trebuie să utilizaţi operatorul **->**. De exemplu, următoarea formă se adresează membrului **bilant**:

```
p->bilant
```

Simbolul -> este denumit uzual *operatorul săgeată* şi este compus din semnul minus urmat de un semn pentru mai mare ca. Săgeata este folosită în locul operatorului punct când aveţi acces la un membru al structurii printr-un pointer la acea structură.

Pentru a vedea cum poate fi utilizat un pointer la o structură, să examinăm acest program simplu, care afişează pe ecran orele, minutele şi secundele folosind un ceas software.

```
/* Afiseaza ora din soft. */
#include <stdio.h>

#define DELAY 128000

struct ora_mea {
    int ora;
    int minute;
    int secunde;
}  ;
void afis(struct ora_mea  *t);
void potrivit(struct ora_mea *t);
void intirzie(void);

void main(void)
{
    struct ora_mea sistora;

    sistora.ora = 0;
    sistora.minute = 0;
    sistora.secunde = 0;

    for(;;) {
          potrivit(&sistora);
          afis(&sistora);
    }
}

void potrivit(struct ora_mea *t)
{
    t->secunde++;
    if(t->secunde==60) {
          t->secunde = 0;
          t->minute++;
    }

    if(t->minute==60) {
          t->minute = 0;
          t->ore++;
    }

    if(t->ore==24) t->ore = 0;
    intirzie();
```

```
}
void afis(struct ora_mea *t)
{
    printf("%02d:", t->ore);
    printf("%02d:", t->minute);
    printf("%02d\n", t->secunde);
void intirzie(void)
{
    long int t;

    /* modificati aceasta dupa cum este necesar */
    for(t=1; t<DELAY; ++t);
}
```

În acest program ora este potrivită prin modificarea definiției lui **DELAY**.

După cum puteți vedea, este definită o structură globală numită **ora_mea**. În interiorul funcției **main()** variabila de tip structură **sistora** este declarată și inițializată cu 00:00:00. Aceasta înseamnă că **sistora** este cunoscută direct doar de funcția **main()**.

Funcțiilor **potrivit,** care modifică ora, și **afis()**, care afișează ora, le este pasată adresa structurii **sistora**. Argumentele ambelor funcții sunt declarate ca pointeri spre o structură de tip **ora_mea**.

În **potrivit()** și **afis()** fiecare membru din **sistora** este accesat printr-un pointer. Deoarece **potrivit()** trimite un pointer către structura **sistora**, poate să potrivească valoarea noii ore. De exemplu, pentru a readuce ora din nou la 0 când se ajunge la 24:00:00, **potrivit** conține această linie de cod:

```
if(t->ora==24) t->ora = 0)
```

Aceasta îi spune compilatorului ca, pentru a readuce **ora** la 0, să folosească adresa păstrată în **t**, care indică spre **sistora** din **main()**.



***REȚINEȚI:** Folosiți operatorul punct pentru a avea acces la elementele structurii când operați asupra structurii însăși. Când aveți un pointer spre o structură, folosiți operatorul săgeată.*

Matrice și structuri în interiorul structurilor

Un membru al unei structuri poate să fie simplu sau de tip compus. Un membru simplu este unul din tipurile de date de bază, așa cum este **int** sau **char**. Ați văzut deja un tip element compus: matricea de caractere folosită în **adrese**. Alte tipuri de date compuse includ matrice uni- și multidimensionale de alte tipuri de date și structuri.

Un membru al unei structuri care este de tip matrice este tratat așa cum ați remarcat ca în exemplele anterioare. De exemplu, să luăm structura:

```
struct x {
    int a[10][10]; /* matrice de intregi 10 x 10 */
    float b;
}  y;
```

Pentru a vă referi la întregul 3,7 al lui **a** din structura **y** scrieți:

```
y.a[3][7]
```

Când o structură este membru al altei structuri, ea poartă denumirea de *structură imbricată*. În următorul exemplu, structura **adrese** este imbricată în **angaj**:

```
struct angaj {
    struct adr adrese; /* structura imbricata */
    float plata;
}   lucrator;
```

Aici, structura **angaj** a fost definită ca având doi membri. Primul este o structură de tipul **adr**, care conține adresa angajatului. Celălalt este **plata**, care conține salariul angajatului. Următorul fragment de cod atribuie 93456 elementului **cod** din **adrese**.

```
lucrator.adrese.cod = 93456;
```

După cum puteți vedea, referirea la membrii fiecărei structuri se face din exterior spre interior. Standardul ANSI C specifică faptul că structurile trebuie să permită imbricarea pe cel puțin 15 niveluri. Propunerea de standard pentru ANSI C++ sugerează să fie permise cel puțin 256 de niveluri.

Câmpuri de biți

Spre deosebire de majoritatea altor limbaje, C posedă o caracteristică intrinsecă denumită *câmp de biți* care permite accesul la un singur bit. Câmpurile de biți pot fi utile din mai multe motive:

- Dacă memoria este limitată, puteți să stocați mai multe variabile *Booleene* (adevărat / fals) într-un singur octet.
- Anumite echipamente transmit prin octeți informații codificate.
- Anumite rutine de criptare trebuie să aibă acces la biții dintr-un octet.

Chiar dacă aceste sarcini pot fi efectuate folosind operatorii pentru biţi, un câmp de biţi poate adăuga mai multă structurare (posibil şi eficienţă) codului dvs.

Pentru a avea acces la biţi, C foloseşte o metodă bazată pe structură. De fapt, un câmp de biţi este efectiv chiar un tip special de membru al unei structuri care defineşte cât de lung trebuie să fie câmpul, în biţi. Forma generală a definirii unui câmp de biţi este:

struct *nume_generic {*
tip nume1 : lungime;
tip nume2 : lungime;
.
.
.
tip numeN : lungime;
} *listă_variabile;*

Aici, *tip* specifică tipul câmpului de biţi, care trebuie să fie de tip **int**, **unsigned**, sau **signed**. Câmpul de biţi cu lungimea 1 trebuie să fie declarat ca fiind **unsigned** deoarece un singur bit nu poate avea semn. (Unele compilatoare permit doar câmpuri de biţi **unsigned**.) Numărul de biţi dintr-un câmp este specificat prin *lungime*.

Câmpurile de biţi sunt utilizate frecvent pentru analizarea intrării de la un echipament hard. De exemplu, portul de stare al unui adaptor de comunicaţie serială poate să returneze un octet de stare organizat astfel:

Bit	Semnificaţie (dacă bitul are valoarea 1)
0	Modificare în linia „clear-to-send"
1	Modificare în „data-set-ready"
2	Detectare de front crescător
3	Modificare în linia de recepţie
4	„Clear-to-send" (CTS)
5	„Data-set-ready" (DST)
6	Apel telefonic
7	Semnal recepţionat

Puteţi să reprezentaţi informaţia dintr-un octet de stare folosind următorul câmp de date:

```
struct tip_stare {
    unsigned delta_cts: 1;
    unsigned delta_dsr: 1;
    unsigned tr_edge:   1;
    unsigned delta_rec: 1;
    unsigned cts:       1;
    unsigned dsr:       1;
    unsigned ring:      1;
    unsigned rec_line:  1;
}   stare;
```

Puteţi să folosiţi o rutină similară cu aceasta pentru a determina când un program să primească sau să transmită date:

```
stare = preia_stare_port();
if(stare.cts) printf("liber pentru transmis");
if(stare.dsr) printf("date pregatite");
```

Pentru a atribui o valoare unui câmp de biţi, utilizaţi forma pe care aţi folosi-o pentru oricare tip de element al structurii. De exemplu, acest fragment de cod şterge câmpul **ring**:

```
stare.ring = 0;
```

După cum puteţi vedea din acest exemplu, accesul la fiecare element se face cu operatorul punct. Dar, dacă accesul la structură se face printr-un pointer, trebuie să folosiţi operatorul ->.

Nu este necesar să numiţi fiecare câmp de biţi. Aceasta determină un acces mai uşor la bitul pe care îl doriţi, trecând peste cei neîntrebuinţaţi. De exemplu, dacă doriţi doar biţii **cts** şi **dsr**, puteţi să declaraţi astfel structura **tip_stare**:

```
struct tip_stare {
    unsigned:       4;
    unsigned cts:   1;
    unsigned dsr:   1;
}   stare;
```

Reţineţi, de asemenea, că biţii de după **dsr** nu au nevoie să fie specificaţi dacă nu sunt folosiţi.

Este corect să amestecaţi membrii normali ai structurii cu câmpuri de biţi. De exemplu,

```
struct angajat {
    struct adr adrese;
    float plata;
    unsigned activ:   1/* activ sau intrerupt */
    unsigned orar:    1/* plata orara */
    unsigned impozit: 3/* impozit rezultat */
};
```

defineşte o înregistrare despre salariat care foloseşte doar un octet pentru a păstra trei informaţii: statutul angajatului, dacă este salariat şi impozitul. Fără câmpul de biţi, aceste informaţii ar fi ocupat trei octeţi.

Variabilele de tip câmp de biţi au anumite restricţii. Nu puteţi să obţineţi adresa unui câmp de biţi. Ele nu pot fi introduse în matrice. Nu puteţi şti dacă aceste câmpuri vor fi rulate de la dreapta la stânga sau de la stânga la dreapta, aceasta diferind de la echipament la echipament. Cu alte cuvinte, orice cod care foloseşte câmpul de biţi poate avea unele caracteristici dependente de echipament.

Uniuni

O *uniune* este o locaţie de memorie care este împărţită în momente diferite între două sau mai multe variabile diferite, în general de tipuri diferite. Declararea unei **uniuni** este similară cu declararea unei structuri. Forma ei generală este:

union *nume_generic {*
tip nume_variabilă;
tip nume_variabilă;
tip nume_variabilă;
.
.
.
} variabile_uniuni;

Iată un exemplu:

```
union tip_u {
    int i;
    char ch;
};
```

Această declarare nu creează nici o variabilă. Acestea se declară ori prin plasarea unui nume la sfârşitul declarării, ori folosind separat instrucţiuni de declarare. Pentru a declara variabila de tip **uniune** cu numele **cnvt** de tip **tip_u** folosind definiţia de mai înainte, scrieţi:

```
union tip_u cnvt;
```

În **cnvt**, atât întregul **i** cât şi caracterul **ch** împart aceeaşi locaţie de memorie. (Desigur, **i** ocupă doi octeţi iar **ch** doar unul.) **Figura 7-2** arată cum **i** şi **ch** împart aceeaşi adresă. Puteţi să vă referiţi la datele stocate în **cnvt** ca la un întreg sau ca la un caracter, din orice punct al programului.

Când este declarată o variabilă de tip **union**, compilatorul alocă automat memorie suficientă pentru a păstra cel mai mare membru al acesteia. De exemplu (presupunând întregii de 2 octeţi), **cnvt** are lungimea de 2 octeţi, astfel încât el poate să îl păstreze pe **i**, chiar dacă **ch** necesită doar un octet.

Pentru a avea acces la membrii unei uniuni, folosiţi aceeaşi sintaxă pe care aţi folosi-o pentru structuri: operatorii punct şi săgeată. Dacă lucraţi direct cu uniunea, folosiţi operatorul punct. Dacă accesul la **union** se face prin pointeri, utilizaţi operatorul săgeată. De exemplu, pentru a atribui valoarea întreagă 10 elementului **i** din **cnvt**, scrieţi:

```
cnvt.i = 10;
```

În exemplul următor, unei funcţii îi este transmis un pointer la **cnvt**:

```
void func1(union tip_u *un)
{
    un->i = 10; /* atribuie 10 lui cnvt folosind functia */
}
```

Utilizarea unei uniuni poate să ajute la crearea de coduri independente de tipul echipamentului (portabile). Deoarece compilatorul ţine seama de mărimea efectivă a membrilor din **union**, nu va exista dependenţă faţă de echipament. Aceasta înseamnă că nu trebuie să vă temeţi de mărimile pentru **int**, **long**, **float** sau oricare alta.

Uniunile sunt folosite frecvent când sunt necesari specificatori speciali de conversie deoarece vă puteţi referi la datele din **uniune** în moduri fundamental diferite. De exemplu, puteţi utiliza o **union** pentru a manevra biţii care formează o dată **double**, pentru a-i modifica precizia sau pentru a face anumite rotunjiri atipice.

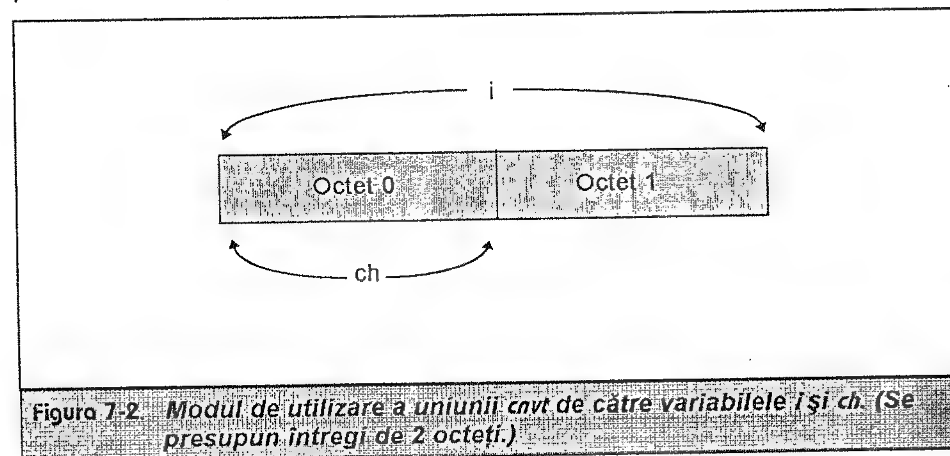


Figura 7-2 *Modul de utilizare a uniunii cnvt de către variabilele i şi ch. (Se presupun întregi de 2 octeţi.)*

Pentru a avea o idee de utilitatea unei **union** când sunt necesari specificatori de tip nestandardizați, să considerăm problema scrierii unui întreg într-un fișier de pe disc. Biblioteca standard de C/C++ nu definește nici o funcție creată special pentru a rezolva această problemă. Deși puteți să scrieți într-un fișier de pe disc orice tip de date (inclusiv una întreagă) folosind **fwrite()**, aceasta este ucigătoare pentru o astfel de operație simplă. Dar, folosind o **union** puteți să creați ușor o funcție numită **scrie()** care scrie reprezentarea binară a unui întreg fișier, câte un octet la fiecare pas. Pentru a vedea cum, mai întâi creați o uniune constând dintr-un întreg și o matrice de caractere cu doi octeți:

```
union csr {
    int i;
    char ch[2];
};
```

Acum puteți folosi **scr** pentru a crea versiunea funcției **scrie()** prezentată în următorul program:

```
#include <stdio.h>
union scr {
    int i;
    char ch[2];
};

scrie(int num, FILE *fp);

void main(void)
{
FILE *fp;

fp = fopen("test.tmp", "w+");

scrie(1000, fp); /* scrie valoarea 1000 ca un intreg */
fclose(fp);
}

scrie(int num, FILE *fp)
{
    union scr cuvint;

    cuvint.i = num;

    putc(cuvint.ch[0], fp); /* scrie prima jumatate */
```



```
    return putc(cuvint.ch[1], fp); /* scrie a doua jumatate */
}
```

Chiar dacă **scrie()** este apelat cu un întreg, ea poate totuși să folosească funcția standard **putc()** pentru a scrie pe rând fiecare octet al întreguluilui **int** într-un fișier de pe disc.



NOTĂ: C++ admite un tip special de uniune numit uniune anonimă, care este discutată în Partea a doua a acestei cărți.

Enumerări

O *enumerare* este un set de constante de tip întreg care specifică toate valorile permise pe care le poate avea o variabilă de acel tip. Enumerările sunt uzuale în viața cotidiană. Iată de exemplu o enumerare a monedelor aflate în circulație în Statele Unite:

penny, nickel (5 cenți), dime (10 cenți), quarter (25 de cenți), jumătate de dolar, dolar

Enumerările sunt definite asemănător structurilor; cuvântul-cheie **enum** semnalează începutul unei enumerări. Forma generală a unei enumerări este:

enum *nume_generic { lista enumerărilor } listă_variabile;*

Aici, atât numele generic al enumerării, cât și lista de variabile sunt opționale. Ca și pentru structuri, numele generic al enumerării este folosit pentru a declara variabile de acel tip. Următorul fragment de cod definește o enumerare numită **monede** și declară **bani** ca fiind de acest tip:

```
enum monede { penny, nickel, dime, quarter,
              jumatate_dolar, dolar};
enum monede bani;
```

Dându-se această declarare, următoarele tipuri de instrucțiuni sunt corecte:

```
bani = dime;
if(bani==quarter) printf("Banul este un quarter.\n");
```

Cheia înțelegerii enumerării este aceea că fiecare dintre simboluri ține locul unei valori întregi. Astfel fiind, ele pot fi folosite oriunde poate fi utilizat și un întreg. Fiecărui simbol i se dă o valoare cu o unitate mai mare decât a

precedentului. Valoarea primului simbol al enumerării este 0. De aceea,

```
printf("%d %d", penny, dime);
```

afişează pe ecran **0 2**.

Puteţi specifica valoarea unuia sau mai multor simboluri folosind o iniţializare. Faceţi aceasta punând după simbol semnul egal şi o valoare întreagă. Simbolurilor care apar după iniţializare li se atribuie valori mai mari decât valoarea de iniţializare precedentă. De exemplu, următorul cod atribuie valoarea 100 lui **quarter**.

```
enum monede { penny, nickel, dime, quarter=100,
              jumatate_dolar, dolar };
```

Acum, valorile acestor simboluri sunt:

penny	0
nickel	1
dime	2
quarter	100
jumatate_dolar	101
dolar	102

O părere uzuală dar greşită despre enumerări este că simbolurile pot fi introduse şi obţinute direct. Nu este cazul. De exemplu, următorul fragment de cod nu va efectua ceea ce se doreşte.

```
/* asa nu va lucra */
bani = dolar;
printf("%s", bani);
```

Amintiţi-vă că **dolar** este un simplu nume al unui întreg; el nu este un şir. Din acelaşi motiv nu puteţi folosi acest cod pentru a obţine rezultatul dorit:

```
/* acest cod este gresit */
strcpy(bani, "dime");
```

Asta înseamnă că un şir care conţine numele unui simbol nu este convertit automat în acel simbol.

De fapt, crearea codului pentru intrarea şi ieşirea simbolurilor enumerării este de-a dreptul plictisitoare (doar dacă nu doriţi să lucraţi cu valorile lor întregi). De exemplu, aveţi nevoie de acest cod pentru a afişa în cuvinte tipul de monede pe care îl conţine **bani**.


```
switch(bani) {
    case penny: printf("penny");
          break;
    case nickel: printf("nickel");
          break;
    case dime: printf("dime");
          break;
    case quarter: printf("quarter");
          break;
    case jumatate_dolar: printf("jumatate_dolar");
          break;
    case dolar: printf("dolar");
}
```

Uneori puteţi să declaraţi o matrice de şiruri şi să folosiţi valoarea enumerării ca pe un indice pentru a transforma acea valoare în şirul corespunzător:

```
char nume[][15]={
    "penny",
    "nickel",
    "dime",
    "quarter',
    "jumatate_dolar",
    "dolar"
};
printf("%s", nume[bani]);
```

Desigur, aceasta lucrează doar dacă nu este iniţializat nici un simbol, deoarece matricea de şiruri trebuie să aibă indici începând cu 0.

Deoarece valorile enumeraţiei trebuie să fie convertite pentru operaţii de I/O manual în valori de şir lizibile de către om, ele sunt foarte utile în rutine care nu efectuează asemenea conversii. De exemplu, o enumerare este utilizată deseori pentru a defini o tabelă de simboluri pentru compilator. Ele mai sunt folosite pentru a dovedi validitatea unui program efectuând un control al redundanţei în timpul compilării care confirmă dacă unei variabile i s-au atribuit doar valori corecte.

Utilizarea lui sizeof pentru asigurarea portabilităţii

Aţi văzut că structurile şi uniunile pot fi folosite pentru a crea variabile de diferite mărimi şi că mărimea efectivă a acestor variabile poate să se modifice de la echipament la echipament. Operatorul **sizeof** calculează mărimea oricărei

variabile sau tip şi poate să elimine dependenţa de echipament a codului programelor dvs. Acest operator este util în special în ceea ce priveşte structurile şi uniunile.

Pentru discuţia care urmează să presupunem o implementare comună multor compilatoare de C/C++ care are următoarele mărimi pentru date:

Tip	Mărimea în octeţi
char	1
int	2
float	4

De aceea, următorul cod va afişa pe ecran numerele 1, 2 şi 4.

```
char ch;
int i;
float f;

printf("%d", sizeof(ch));
printf("%d", sizeof(i));
printf("%d", sizeof(f));
```

Mărimea unei structuri este egală *sau mai mare ca* suma mărimilor membrilor săi, ca în exemplul:

```
struct s {
    char ch;
    int i;
    float f;
} var_s;
```

Aici, **sizeof(var_s)** este cel puţin 7 (4+2+1). Însă, mărimea lui **var_s** poate fi şi mai mare deoarece compilatorului îi este permis să completeze cu spaţii o structură pentru a realiza alinierea cuvântului sau a paragrafului. (Un paragraf are 16 octeţi.) De vreme ce mărimea unei structuri poate fi mai mare decât suma mărimilor membrilor săi, de câte ori doriţi să ştiţi mărimea ei trebuie să folosiţi **sizeof**.

Deoarece **sizeof** este un operator cu acţiune în timpul compilării, toate informaţiile necesare calculării mărimii oricărei variabile sunt cunoscute abia în timpul acestui proces. Aceasta are importanţă în special pentru **union**, deoarece mărimea unei uniuni este întotdeauna egală cu mărimea celui mai mare membru al său. De exemplu, avem uniunea:

```
union u {
    char ch;
    int i;
    float f;
} var_u;
```

Aici, **sizeof(var_u)** este 4. Nu are importanţă ce conţine efectiv **var_u** în timpul rulării. Tot ce contează este mărimea celui mai mare membru, deoarece orice **union** este la fel de mare cât elementul său cel mai mare.

typedef

C permite să definiţi explicit noi nume de tipuri de date prin utilizarea cuvântului-cheie **typedef**. De fapt nu *creaţi* efectiv un nou tip de date, ci mai degrabă definiţi un nou nume pentru un tip existent. Acest proces poate să ajute să faceţi mai portabile programele dependente de echipament. Dacă definiţi propriul dvs. nume de tip pentru fiecare tip de date din program care depinde de echipament, atunci, la compilarea într-un mediu nou, vor trebui modificate doar instrucţiunile **typedef**. De asemenea, **typedef** vă poate ajuta pentru documentaţia codului permiţând nume descriptive pentru tipurile de date standard. Forma generală a instrucţiunii **typedef** este:

typedef *tip numenou;*

unde *tip* este orice tip de date valid iar *numenou* este un nou nume pentru acest tip. Noul nume pe care îl definiţi este unul în plus, nu o înlocuire a celui existent.

De exemplu, puteţi crea un nou nume pentru **float** utilizând:

```
typedef float bilant;
```

Această instrucţiune spune compilatorului să recunoască **bilant** ca un alt nume pentru **float**. Apoi, puteţi crea o variabilă de tip **float** utilizând **bilant**:

```
bilant scadent;
```

Aici, **scadent** este o variabilă în virgulă mobilă de tip **bilant** care este un alt cuvânt pentru **float**.

Acum, o dată ce am definit **bilant**, el poate fi folosit în alt **typedef**. De exemplu,

```
typedef bilant neacoperit;
```

spune compilatorului să recunoască **neacoperit** ca un alt nume pentru **bilant**, care este un alt nume pentru **float**.

Folosind **typedef** puteţi să faceţi codul dvs. mai uşor de citit şi de introdus pe un alt echipament. Dar reţineţi, nu creaţi nici un tip nou de date.

Capitolul 8

I/O la consolă



Capitolul acesta şi următorul prezintă sistemul I/O (intrare/ieşire) al limbajului C. În C intrările şi ieşirile sunt efectuate de funcţiile de bibliotecă. Sistemul C de I/O este o metodă tehnică elegantă care oferă un mecanism flexibil, dar coerent de transfer al datelor între echipamente. Totuşi el este destul de mare şi implică mai multe funcţii.

C admite atât I/O de la consolă cât şi prin fişiere, nefăcând practic o deosebire cele două procese. Totuşi, din punct de vedere conceptual, ele reprezintă două lumi diferite. Acest capitol examinează în detaliu funcţiile de I/O pentru consolă. Următorul capitol prezintă sistemul de I/O pentru fişiere şi descrie legătura între cele două sisteme.

Acest capitol acoperă, cu o singură excepţie, doar funcţiile de I/O pentru consolă definite de standardul ANSI C. Acest standard nu defineşte nici o funcţie care să efectueze diverse operaţii de control al ecranului (cum ar fi poziţionarea cursorului) sau care să afişeze grafică, deoarece operaţiile respective diferă foarte mult între echipamente. În schimb, funcţiile standard pentru consolă din C efectuează doar operaţii pentru ieşiri de tip terminal (TTY). Cu toate acestea, majoritatea compilatoarelor includ în bibliotecile lor atât funcţii de control al ecranului cât şi funcţii pentru grafică ce se aplică mediului specific în care este proiectat să ruleze compilatorul. (Pentru funcţiile nestandardizate de I/O trebuie să consultaţi manualul utilizatorului.)

Acest capitol se referă la funcţiile de I/O pentru consolă care preiau intrări de la tastatură şi produc ieşire pe ecran. Totuşi, pentru acţiunile de I/O, aceste funcţii au efectiv ca sursă şi/sau destinaţie intrările şi ieşirile standard în/din sistem. Mai mult, intrările şi ieşirile standard pot fi redirecţionate către alte echipamente. Aceste concepte sunt explicate în **Capitolul 9**.

O notă cu importanţă practică

Partea întâi a acestei cărţi foloseşte sistemul de I/O definit de limbajul C. Chiar dacă C++ admite în întregime funcţiile de I/O din C, el defineşte propriul său sistem de I/O orientat pe obiecte. De aceea, dacă scrieţi programe orientate pe obiecte, veţi prefera să folosiţi sistemul de I/O specific lui C++, nu sistemul ANSI C descris în acest capitol. Sistemul de I/O al limbajului C este discutat în această carte din următoarele motive:

- În următorii ani C şi C++ vor coexista; de asemenea, multe programe vor fi hibride, conţinând atât cod C cât şi C++. În plus, va fi uzuală dezvoltarea programelor din C în programe în C++. Pentru aceasta vor fi necesare atât cunoştinţe despre sistemul de I/O din C cât şi despre cel din C++. De exemplu, pentru a modifica funcţiile din C pentru I/O în funcţii de I/O orientate pe obiecte, va trebui să ştiţi cum operează ambele sisteme.
- Înţelegerea principiilor de bază ale sistemelor de I/O din C este determinantă pentru înţelegerea sistemului din C++, orientat pe obiecte. (Amândouă operează cu aceleaşi concepte generale.)
- În anumite situaţii (de exemplu, în programele foarte scurte), poate să fie mai uşor să folosiţi abordarea de I/O neorientată pe obiecte din C, decât cea orientată pe obiecte definită de C++.

În plus, există o regulă nescrisă, ca orice programator în C++ să fie şi programator în C. Dacă nu veţi şti să utilizaţi sistemul de I/O din C orizontul dvs. profesional va fi limitat.

Citirea şi scrierea caracterelor

Cele mai simple funcţii de I/O pentru consolă sunt **getchar()**, care citeşte un caracter de la tastatură şi **putchar()**, care scrie un caracter pe ecran. Funcţia **getchar()** aşteaptă până este apăsată o tastă şi returnează valoarea sa. De asemenea, tasta apăsată are automat ecou pe ecran. Funcţia **putchar()** scrie un caracter pe ecran în poziţia curentă a cursorului. Iată prototipurile pentru **getchar** şi **putchar()**:

```
int getchar(void);
int putchar(int c);
```

Fişierul antet pentru aceste funcţii este STDIO.H. După cum arată prototipul, funcţia **getchar()** este declarată ca returnând un întreg. Totuşi, puteţi atribui această valoare unei variabile de tip **char**, aşa cum se face uzual, deoarece caracterul este conţinut în octetul de ordin inferior. (Octetul de ordin superior este de obicei 0.) Dacă apare o eroare, **getchar()** returnează **EOF**.

În cazul lui **putchar()**, chiar dacă este declarată ca preluând un parametru de tip întreg, de obicei o veţi apela folosind un argument de tip caracter. Ieşirea pe ecran constă efectiv doar din octetul său de ordin inferior. Funcţia **putchar()** returnează caracterul scris sau, în cazul apariţiei unei erori, **EOF**. (Funcţia macro **EOF** este definită în STDIO.H şi, în general, este egală cu -1.)

Următorul program ilustrează **getchar()** şi **putchar()**. El introduce caractere de la tastatură şi le afişează după ce înlocuieşte literele mari cu cele mici şi reciproc. Pentru a încheia programul, se introduce un punct.

```
#include <stdio.h>
#include <ctype.h>

void main(void)
{
    char ch;
```

```
    printf("Introduceti un text (tastati un punct pentru
          a iesi din program).\n");
    do {
          ch = getchar();
          if(islower(ch)) ch = toupper(ch);
          else ch = tolower(ch);
          putchar(ch);
    }     while (ch != '.');
}
```

O problemă cu getchar()

Există unele probleme potențiale cu **getchar()**. Standardul ANSI C o definește ca fiind compatibilă cu originalul, versiunea de C bazată pe UNIX. Din nefericire, în forma sa inițială **getchar** introduce intrarea în buffer (memoria tampon) până când este apăsat ENTER. Aceasta este denumită intrare de tip *line-buffered* și a fost metoda folosită în sistemul original UNIX; trebuie să tastați ENTER înainte ca ceea ce ați scris să fie transmis efectiv programului. De asemenea, deoarece **getchar()** introduce doar un caracter la fiecare apelare, line-buffering poate să lase unul sau mai multe caractere să aștepte la rând, lucru enervant în mediile interactive. Chiar dacă standardul ANSI C specifică **getchar()** ca putând fi implementată ca funcție interactivă, rareori este așa. De aceea, dacă programul anterior nu s-a comportat cum vă așteptați, acum știți de ce.

Alternative la getchar()

Se poate ca **getchar()** să nu fie implementat de compilatorul dvs. astfel încât să fie util într-un mediu interactiv. Dacă este așa, probabil veți dori să folosiți alte funcții pentru a citi caractere de la tastatură. Standardul ANSI C nu definește nici o funcție care să garanteze oferirea unei intrări interactive, dar teoretic toate compilatoarele au așa ceva. Chiar dacă aceste funcții nu sunt definite de ANSI, ele sunt folosite uzual deoarece **getchar()** nu acoperă cerințele majorității programatorilor.

Două dintre cele mai folosite funcții alternativă, **getch()** și **getche()**, au următoarele prototipuri:

```
int getch(void);
int getche(void);
```

Pentru majoritatea compilatoarelor, prototipurile acestor funcții se găsesc în CONIO.H. Funcția **getch()** așteaptă apăsarea unei taste după care returnează imediat. Ea nu are ecou pe ecran. Funcția **getche()** este identică cu **getch()**, dar caracterul apare pe ecran. Această carte utilizează **getch()** sau **getche()** în loc de **getchar()** atunci când caracterul trebuie să fie citit de la tastatură într-un program interactiv. Totuși, în cazul în care programul dvs. nu admite aceste funcții alternative, sau dacă **getchar()** este implementat de către compilatorul dvs. ca funcție interactivă, trebuie să-l folosiți pe **getchar()**.

Următorul exemplu înlocuiește în programul prezentat anterior **getchar()** cu **getch()**:

```
#include <stdio.h>
#include <conio.h>
#include <ctype.h>

void main(void)
{
   char ch;

   printf("Introduceti un text (tastati un punct pentru
          a iesi din program).\n");

   do {
          ch = getch();

          if(islower(ch)) ch = toupper(ch);
          else ch = tolower(ch);
          putchar(ch);
   } while(ch != ".");
}
```

Citirea și scrierea șirurilor

Următorul pas în I/O pentru consolă, din punctul de vedere al complexității și al forței, sunt funcțiile **gets()** și **puts()**. Ele vă permit citirea și scrierea șirurilor de caractere de la/la consolă.

Funcția **gets()** citește un șir de caractere introduse de la tastatură și le plasează la adresa indicată de argumentul său. Puteți scrie caractere de la tastatură până când apăsați un caracter de linie nouă. Acesta nu face parte din șir; în locul său este plasat la sfârșitul șirului un caracter de încheiere null și apoi **gets()** se încheie. De fapt, nu puteți folosi **gets()** pentru a returna un caracter de linie nouă (chiar dacă **getchar()** o poate face). Puteți să corectați greșelile de tastare folosind backspace (șterge un caracter înapoi) înainte de a apăsa ENTER. Prototipul funcției **gets()** este:

char *gets(char *sir);

unde *sir* este o matrice care primeşte caracterele introduse de către utilizator. De asemenea, **gets()** returnează *sir*. Prototipul lui **gets()** se găseşte în STDIO.H. Următorul program citeşte un şir dintr-o matrice **sir** şi îi afişează lungimea.

```
#include <stdio.h>
#include <string.h>

void main(void)
{
    char sir[80];

    gets(sir);
    printf("Lungimea este %d", strlen(sir));
}
```

Funcţia **puts()** scrie pe ecran argumentul său şir, urmat de o linie nouă. Prototipul său este:

int puts(const char **sir*);

puts() admite aceleaşi coduri backslash ca şi **printf()**, cum ar fi '\t' pentru spaţiul de tabulare. O apelare a lui **puts()** necesită mult mai puţine resurse de sistem decât aceeaşi apelare pentru **printf()**, deoarece **puts()** poate să emită ca ieşire doar un şir de caractere - nu poate afişa numere sau efectua conversii de formatare. De aceea, **puts()** ocupă un spaţiu mai mic şi rulează mai repede decât **printf()**. Din acest motiv, funcţia **puts()** este folosită deseori când este necesar să avem un cod extrem de optimizat. În cazul apariţiei unei erori, funcţia **puts()** returnează **EOF**. Altfel, returnează o valoare non-negativă. Totuşi, când scrieţi la o consolă, puteţi presupune că în mod normal nu va apărea nici o eroare, deci valoarea returnată de **puts()** este rareori urmărită. Următoarea instrucţiune afişează **hello**.

```
puts("hello");
```

Tabelul 8-1 rezumă funcţiile de bază de I/O pentru consolă.

Următorul program, un simplu dicţionar, ilustrează mai multe funcţii de bază de I/O pentru consolă. El solicită utilizatorului să introducă un cuvânt şi apoi verifică dacă acesta coincide cu unul din baza proprie de date. În caz afirmativ, programul afişează semnificaţia cuvântului. Fiţi atent mai ales la indirectarea folosită în acest program. Dacă aveţi vreo problemă de a-l înţelege, amintiţi-vă că matricea **dic** este o matrice de pointeri către şiruri. Reţineţi că lista va trebui să se termine cu două caractere null.

Funcţia	**Operaţia**
getchar()	Citeşte un caracter de la tastatură; aşteaptă caracter de linie nouă.
getche()	Citeşte un caracter şi are ecou; nu necesită caracter de linie nouă; nu este definit de standardul ANSI C, dar este o extindere uzuală.
getch()	Citeşte un caracter fără ecou; nu necesită caracter de linie nouă; nu este definit de standardul ANSI C, dar este o extindere uzuală.
putchar()	Scrie un caracter pe ecran.
gets()	Citeşte un şir de la tastatură.
puts()	Scrie un şir pe ecran.

Tabelul 8-1. *Funcţiile I/O de bază*

```
/* Un dictionar simplu */
#include <stdio.h>
#include <conio.h>
#include <string.h>
#include <ctype.h>

/* lista de cuvinte si semnificatii */
char *dic[][40] = {
    "atlas", "o culegere de harti",
    "masina", "un vehicol motorizat",
    "telefon", "un echipament de comunicare",
    "avion", "o masina zburatoare",
    "", "" /* lista se termina cu null */
};
void main(void)
{
    char cuvint[80], ch;
    char **p;

    do {
         puts("\nIntroduceti un cuvint: ");
         gets(cuvint);

         p = (char **)dic;

         /* gaseste cuvintul si ii afiseaza semnificatia */
```

```
            do {
                  if(!strcmp(*p, cuvint)) {
                        puts("inseamna:");
                        puts(*(p+1));
                        break;
                  }
                  if(!strcmp(*p, cuvint)) break;
                  p = p + 2; /* continua lista */
            } while(*p);
            if(!*p) puts("cuvintul nu este in dictionar");
            printf("altul? (y/n): ");
            ch = getche();
      }     while(toupper(ch) != 'N');
}
```

I/O de la consolă, formatate

Funcţiile **printf()** şi **scanf()** efectuează intrări şi ieşiri formatate - ele pot citi şi respectiv scrie date în diverse formate pe care le puteţi controla dvs. Funcţia **printf()** scrie date pentru consolă. Funcţia **scanf()**, complementul său, citeşte date de la tastatură. Ambele funcţii pot să lucreze cu orice tip de date existent, inclusiv caractere, şiruri şi numere.

printf()

Prototipul lui **printf()** este:

int printf(const char **control_sir, ...*);

Prototipul lui **printf()** se află în STDIO.H. Funcţia returnează numărul de caractere scrise sau, dacă apare o eroare, o valoare negativă.

control_sir constă din două tipuri de simboluri. Primul tip este format din caracterele care vor fi afişate pe ecran. Al doilea conţine specificatorii de format care definesc modul în care sunt afişate argumentele care îi urmează. Un specificator de formatare începe cu un semn pentru procent şi este urmat de un cod de format. Trebuie să existe exact acelaşi număr de argumente ca şi acela al specificatorilor de format, iar specificatorii de format şi argumentele sunt corelaţi în ordine, de la stânga la dreapta. De exemplu, această apelare pentru **printf()**:

```
printf("Imi place %c %s", 'C', "++ foarte mult!");
```

afişează


```
Imi place C++ foarte mult!
```

Funcţia **printf()** acceptă o mare varietate de specificatori de format, aşa cum se arată în **Tabelul 8-2**.

Afişarea caracterelor

Pentru a afişa un caracter individual, utilizaţi **%c**. Aceasta face ca argumentul de căutare să fie afişat nemodificat pe ecran.

Pentru a afişa un şir, folosiţi **%s**.

Afişarea numerelor

Pentru a specifica un număr în baza 10 cu semn folosiţi **%d** sau **%i**. Aceste specificatoare de format sunt echivalente; sunt admise ambele din motive de istoric.

Pentru a afişa o valoare fără semn, folosiţi **%u**.

Specificatorul de format **%f** afişează numere în virgulă mobilă.

Specificatorii **%e** şi **%E** spun lui **printf()** să afişeze un argument de tip **double** în

Cod	Format
%c	Caracter
%d	Numere întregi în baza 10, cu semn
%i	Numere întregi în baza 10, cu semn
%e	Notaţie ştiinţifică (cu litera e mică)
%E	Notaţie ştiinţifică (cu litera E mare)
%f	Număr zecimal în virgulă mobilă
%g	Foloseşte %e sau %f, care din ele este mai mic
%G	Foloseşte %E sau %f, care din ele este mai mic
%o	Număr în octal fără semn
%s	Şir de caractere
%u	Numere întregi zecimale fără semn
%x	Numere hexazecimale fără semn (cu litere mici)
%X	Numere hexazecimale fără semn (cu litere mari)
%p	Afişează un pointer
%n	Argumentul asociat este un pointer de tip întreg în care a fost plasat numărul de caractere scrise până atunci
%%	Afişează un semn %

Tabelul 8-2 *Specificatori de format printf()*

notație științifică. Numerele reprezentate în notație științifică au această formă generală:

x.dddddE+/-yy

Dacă doriți să afișați litera mare „E", folosiți formatul **%E**; altfel utilizați **%e**.

Puteți să îi spuneți lui **printf()** să folosească fie **%f** fie **%e**, utilizând specificatorii de format **%g** sau %G. Aceasta determină ca **printf()** să aleagă specificatorul de format care are cea mai scurtă formă de ieșire. Unde se poate, folosiți **%G** dacă doriți ca „E" să fie scris cu literă mare; altfel, folosiți **%g**. Următorul program arată efectul specificatorului de format **%g**.

```
#include <stdio.h>

void main(void)
{
    double f;

    for(f=1.0; f<1.0e+10; f=f*10)
          printf("%g ", f);
}
```

El produce următoarea ieșire:

```
1 10 100 1000 10000 100000 1e+06 1e+07 1e+08 1e+09
```

Puteți afișa întregii fără semn în format octal sau hexazecimal folosind **%o** și respectiv **%x**. Deoarece numerele în hexazecimal folosesc literele de la A la F pentru a reprezenta valorile de la 10 la 15, puteți să le afișați cu literă mare sau mică. Pentru literă mare, utilizați specificatorul de format **%X**; pentru litere mici, folosiți **%x**, așa cum este prezentat aici:

```
#include <stdio.h>

void main(void)
{
    unsigned num;

    for(num=0; num<255; num++) {
          printf("%o ", num);
          printf("%x ", num);
          printf("%X\n", num);
    }
}
```


Afișarea unei adrese

Dacă doriți să afișați o adresă, folosiți **%p**. Acest specificator de format determină ca **printf()** să afișeze o adresă într-un format compatibil cu modul de adresare utilizat de calculator. Următorul program afișează adresa lui **exemplu**:

```
#include <stdio.h>

int exemplu;

void main(void)
{
    printf("%p", &exemplu);
}
```

Specificatorul %n

Specificatorul de format **%n** este diferit de ceilalți. În loc să îi spună lui **printf()** să afișeze ceva, el face ca acesta să încarce variabila spre care indică argumentul corespunzător cu o valoare egală cu numărul de caractere care au fost afișate. Cu alte cuvinte, valoarea care corespunde specificatorului de format **%n** trebuie să fie un pointer la o variabilă. După returnarea apelării lui **printf()**, variabila va păstra numărul de caractere de ieșire, până în momentul în care a fost întâlnit **%n**. Examinați programul următor pentru a înțelege acest cod de formare cumva mai puțin obișnuit:

```
#include <stdio.h>

void main(void)
{
    int numara;

    printf("acesta%n este un test\n", &numara);
    printf("%d", numara);
}
```

Acest program afișează **acesta este un test** urmat de numărul 6. Specificatorul de format **%n** este folosit în primul rând pentru a permite programului să efectueze o formatare dinamică.

Modelatori de format

Mulți specificatori de format pot accepta modelatori care modifică ușor semnificația lor. De exemplu, puteți specifica un minim de mărime a câmpului, numărul de cifre zecimale și alinierea la stânga. Modelatorul de format se află între semnul procent și codul pentru format. În continuare sunt prezentați acești modelatori.

Specificatorul pentru mărimea minimă a câmpului

Un întreg plasat între semnul % și codul pentru format acționează ca un *specificator pentru mărimea minimă a câmpului.* Acesta umple ieșirea cu spații pentru a se asigura că ajunge la o anumită lungime minimă. Dacă șirul sau numărul este mai mare decât minimul, el va fi afișat complet. Tot ce rămâne liber este umplut cu spații. Dacă doriți să completați cu zero, plasați un 0 înaintea specificatorului de mărime pentru câmp. De exemplu, **%05** va umple cu zero un număr mai mic de cinci cifre, astfel încât lungimea sa totală să fie cinci. Următorul program exemplifică specificatorul pentru câmp minim.

```
#include <stdio.h>

void main(void)
{
    double numar;

    numar = 10.12304;

    printf("%f\n", numar);
    printf("%10f\n", numar);
    printf("%012f\n", numar);
}
```

Acest program determină următoarea ieșire:

```
10.123040
 10.123040
00010.123040
```

Modelatorul de câmp minim este folosit uzual pentru alinierea coloanelor în tabele. De exemplu, următorul program realizează un tabel pentru pătratele și cuburile numerelor între 1 și 19.

```
#include <stdio.h>
```



```
void main(void)
{
    int i;

    /* afiseaza un tabel de patrate si cuburi de numere */
    for(i=1; i<20; i++)
        printf("%8d %8d %8d\n", i, i*i, i*i*i);
}
```

Iată un exemplu de ieșire a acestui program:

```
       1        1        1
       2        4        8
       3        9       27
       4       16       64
       5       25      125
       6       36      216
       7       49      343
       8       64      512
       9       81      729
      10      100     1000
      11      121     1331
      12      144     1728
      13      169     2197
      14      196     2744
      15      225     3375
      16      256     4096
      17      289     4913
      18      324     5832
      19      361     6859
```

Specificatori de precizie

Specificatorul de precizie urmează specificatorului de câmp minim (dacă există unul). El constă dintr-un punct urmat de un întreg. Semnificația sa exactă depinde de tipul de date căruia i se aplică.

Când aplicați specificatorul de precizie unei date în virgulă mobilă utilizând formatele **%f**, **%e** sau **%E**, el determină numărul de zecimale afișate. De exemplu, **%10.4f** afișează un număr de cel puțin 10 caractere cu patru cifre zecimale. Dacă nu specificați precizia, vor fi folosite implicit șase cifre.

Când specificatorul de precizie se aplică lui **%g** sau **%G**, el indică numărul de cifre semnificative.

Aplicat şirurilor, acest specificator precizează lungimea maximă a câmpului. De exemplu, **%5.7** afişează un şir cu lungimea de cel puţin cinci caractere, dar nu mai mare de şapte. Dacă şirul este mai lung decât mărimea maximă a câmpului, ultimele caractere vor fi eliminate.

Când se aplică tipului de întregi, specificatorul de precizie determină numărul minim de cifre care apar pentru fiecare număr. Pentru a ajunge la numărul cerut de cifre, se adaugă zerouri în faţa sa.

Următorul program ilustrează specificatorul de precizie.

```
#include <stdio.h>

void main(void)
{
    printf("%.4f\n", 123.1234567);
    printf("%3.8d\n", 1000);
    printf("%10.15s\n", "Acesta este un test simplu.");
}
```

El determină următoarea ieşire:

```
123.1235
00001000
Acesta este un
```

Alinierea ieşirilor

Toate ieşirile sunt aliniate implicit la dreapta. Aceasta înseamnă că, dacă un câmp este mai mare decât datele afişate, ele vor fi plasate în capătul din dreapta al câmpului. Puteţi forţa ca datele să fie aliniate la stânga plasând un semn minus imediat după %. De exemplu, **%-10.2f** va alinia la stânga un număr în virgulă mobilă cu două zecimale într-un câmp de 10 caractere.

Următorul program prezintă alinierea la stânga

```
#include <stdio.h>

void main(void)
{
    printf("aliniat la dreapta:%8d\n", 100);
    printf("aliniat la stinga:%-8d\n", 100);
}
```


Manevrarea altor tipuri de date

Există doi modelatori de format care permit lui **printf()** să afişeze întregi de tip **short** şi **long**. Aceşti modelatori pot fi aplicaţi specificatorilor de tip **d**, **i**, **o**, **u** şi **x**. Modelatorul **l** (lirera l mică) spune lui **printf()** că urmează un tip de date **long**. De exemplu, **%ld** înseamnă că va fi afişat un **long int**. Modelatorul **h** determină **printf()** să afişeze un întreg de tip **short**. De exemplu, **%hu** indică un tip de date **short unsigned int**.

Modelatorul **L** se poate afla în faţa specificatorilor în virgulă mobilă **e**, **f** şi **g** indicând că va urma un **long double**.

Modelatorii * şi

Funcţia **printf()** admite doi modelatori în plus pentru unii dintre specificatorii săi de format: * şi #.

Punând # în faţa specificatorilor **g**, **G**, **f**, **E** sau **e**, asiguraţi prezenţa unui punct zecimal chiar dacă nu există cifre zecimale. Dacă îl veţi folosi în faţa specificatorilor de format **x** sau **X**, numărul hexazecimal va fi afişat având în faţă **0x**. În faţa specificatorului **o**, # va determina ca numărul să fie afişat cu un 0 pe prima poziţie. Nu puteţi aplica # nici unui alt specificator de format.

Mărimea minimă a câmpului şi precizia specificatorilor pot fi specificate nu numai prin constante, ci şi prin argumente ale lui **printf()**. Pentru a realiza aceasta, folosiţi un * pentru a rezerva un loc. Când este parcurs şirul formatului, **printf()** va înlocui * cu câte un argument, în ordinea în care care apar acestea. De exemplu, în **Figura 8-1** mărimea minimă a câmpului este 10, precizia este 4 iar valoarea care va fi afişată este **123.3**.

Următorul program ilustrează utilizarea specificatorilor # şi *.

```
#include <stdio.h>

void main(void)
{
    printf("%x %#x\n", 10, 10);
    printf("%*.*f", 10, 4, 1234.34);
}
```

```
printf("%*.*f", 10, 4, 123.3);
```

Figura 8-1 *Înlocuirea lui * cu valoarea sa*

scanf()

scanf() este o rutină de uz general pentru intrări de la consolă. Ea poate să citească toate tipurile de date încorporate şi să facă automat conversia numerelor în formatul intern corect. Se aseamănă mult cu reciproca sa, **printf()**. Prototipul funcţiei **scanf()** este:

int scanf(const char **sir_control*,...);

Prototipul se găseşte în STDIO.H. Funcţia **scanf()** returnează numărul de elemente de date cărora li s-a atribuit cu succes o valoare. Dacă apare o eroare, **scanf()** returnează **EOF**. *sir_control* determină modul de citire a valorilor în variabilele din lista de argumente spre care indică.

Şirul de control constă din trei clasificări ale caracterelor:

- Specificatori de format
- Caractere spaţii libere
- Caractere spaţii ocupate

Să le privim acum pe fiecare.

Cod	Semnificaţie
%c	Citeşte un singur caracter.
%d	Citeşte un număr întreg zecimal.
%i	Citeşte un număr întreg zecimal.
%e	Citeşte un număr în virgulă mobilă.
%f	Citeşte un număr în virgulă mobilă.
%g	Citeşte un număr în virgulă mobilă.
%o	Citeşte un număr în octal.
%s	Citeşte un şir.
%x	Citeşte un număr în hexazecimal.
%p	Citeşte un pointer.
%n	Primeşte o valoare egală cu numărul de caractere citite până atunci.
%u	Citeşte un număr întreg fără semn.
%[ ]	Caută un set de caractere.

Tabelul 8-3 *Specificatorii de format pentru scanf()*

Specificatori de format

Specificatorii pentru formatul de intrare sunt precedaţi de semnul % şi spun funcţiei **scanf()** ce tip de date urmează să fie citite. Aceste coduri sunt prezentate în **Tabelul 8-3**. Specificatorii de format sunt corelaţi cu argumentele din listă, în ordine, de la stânga la dreapta. Să urmărim câteva exemple.

Intrări de numere

Pentru a citi un număr zecimal, folosiţi specificatorii **%d** sau **%i**. (Aceşti specificatori, care fac acelaşi lucru, sunt păstraţi amândoi pentru compatibilitate cu versiunile vechi de C.)

Pentru a citi un număr în virgulă mobilă, reprezentat atât în notaţie standard cât şi în notaţie ştiinţifică, folosiţi **%e**, **%f** sau **%g**. (Din nou, aceşti specificatori, care fac exact acelaşi lucru, sunt introduşi pentru compatibilitate cu versiunile vechi de C.)

Puteţi utiliza **scanf()** pentru a citi numere întregi atât în formă octală cât şi hexazecimală folosind comenzile pentru format **%o** şi respectiv **%x**. **%x** poate să fie scris cu literă mică sau mare. În oricare caz, atunci când introduceţi numere hexazecimale puteţi să scrieţi literele de la A la F mari sau mici. Următorul program citeşte un număr în octal şi unul în hexazecimal.

```
#include <stdio.h>

void main(void)
{
   int i, j;

   scanf("%o%x", &i, &j);
   printf("%o %x", i, j);
}
```

Funcţia **scanf()** încetează citirea unui număr când este întâlnit primul caracter nenumeric.

Intrări de întregi fără semn

Pentru a introduce un întreg fără semn, folosiţi specificatorul de format **%u**. De exemplu,

```
unsigned num;
scanf("%u", &num);
```

citeşte un număr fără semn şi pune valoarea sa în **num**.

Citirea caracterelor individuale folosind scanf()

După cum aţi învăţat mai devreme în acest capitol, puteţi citi caractere individuale folosind **getchar()** sau o funcţie derivată. Puteţi, de asemenea, să întrebuinţaţi în acest scop **scanf()** dacă utilizaţi specificatorul de format **%c**. Totuşi, ca şi majoritatea implementărilor lui **getchar()**, **scanf()** trimite în general intrările în buffer când este folosit specificatorul **%c**. Acest lucru este puţin incomod într-un mediu interactiv.

Chiar dacă spaţiile simple, cele de tabulare şi de început de linie sunt separatoare pentru câmpuri atunci când se citesc alte tipuri de date, când se citeşte un singur caracter spaţiile libere sunt citite ca şi oricare alt caracter. De exemplu, pentru un şir de intrare „**x y**", următorul fragment de cod

```
scanf("%c%c%c", &a, &b, &c);
```

introduce caracterul **x** în **a**, un spaţiu în **b** şi caracterul **y** în **c**.

Citirea şirurilor

Funcţia **scanf()** poate fi folosită pentru a citi şiruri din streamul de intrare utilizând specificatorul de format **%s**. Acesta determină ca **scanf()** să citească toate caracterele până când întâlneşte un caracter de spaţiu liber. Caracterele care sunt citite sunt introduse în matricea de tip caracter spre care indică argumentul corespunzător, iar rezultatul se încheie cu un caracter null. Relativ la **scanf()**, un spaţiu liber poate fi un spaţiu, o linie nouă, un spaţiu de tabulare orizontală sau verticală sau o pagină nouă. Spre deosebire de **gets()**, care citeşte un şir până când apare un caracter de linie nouă, **scanf()** îl citeşte până la introducerea primului spaţiu. Aceasta înseamnă că nu puteţi folosi **scanf()** pentru a citi un astfel de şir: „acesta este un test", deoarece primul spaţiu încheie procesul de citire. Pentru a vedea efectul specificatorului **%s**, încercaţi acest program folosind şirul „va salut".

```
#include <stdio.h>

void main(void)
{
    char sir[80];

    printf("Introduceti un sir: ");
    scanf("%s", sir);
    printf("Iata sirul dvs.: %s", sir);
}
```


Programul va răspunde doar cu zona „va" a şirului.

Introducerea unei adrese

Pentru a introduce o adresă de memorie, folosiţi specificatorul de format **%p**. Acesta determină **scanf()** să citească o adresă în formatul definit de arhitectura CPU (unitatea centrală de prelucrare). De exemplu, următorul program introduce o adresă şi apoi afişează ce se află la acea adresă.

```
#include <stdio.h>

void main(void)
{
    char *p;

    printf("Introduceti o adresa: ");
    scanf("%p", &p);
    printf("Valoarea din locatia %p este %c\n", p, *p);
}
```

Specificatorul %n

Specificatorul **%n** determină **scanf()** să atribuie variabilei spre care indică argumentul corespunzător numărul de caractere citit din streamul de intrare până în punctul în care este întâlnit **%n**.

Utilizarea specificatorului pentru seturi

Funcţia **scanf()** admite un specificator de format pentru uz general numit „scanset". Un *scanset* defineşte un grup de caractere. Când **scanf()** prelucrează un scanset, el va introduce caractere atât timp cât ele fac parte din grupul definit de scanset. Caracterele citite vor fi atribuite matricei de tip caracter spre care indică argumentul care îi corespunde scansetului. Un scanset se defineşte scriind caracterele de citit între paranteze drepte. Paranteza dreaptă de început trebuie să fie precedată de un semn procent. De exemplu, următorul scanset spune lui **scanf()** să citească doar caracterele X, Y şi Z.

```
%[XYZ]
```

Când folosiţi un scanset, **scanf()** continuă să citească şi să pună caractere în matricea de tip caracter corespunzătoare, până când întâlneşte un caracter care nu

se află în scanset. La returnarea lui **scanf()**, această matrice va conține un șir terminat cu un null, care conține caracterele citite. Pentru a vedea cum lucrează aceasta, să încercăm următorul program:

```
#include <stdio.h>

void main(void)
{
    int i;
    char sir[80], sir2[80];

    scanf("%d%[abcdefg]%s", &i, sir, sir2);
    printf("%d %s %s", i, sir, sir2);
}
```

Introduceți **123abcdtye** urmat de ENTER. Programul va afișa **123 abcd tye**. Deoarece „t" nu face parte din scanset, când îl va întâlni, **scanf()** va opri citirea caracterelor în **sir**. Caracterele rămase vor fi trecute în **sir2**.

Puteți să specificați reciproca unui set punând drept prim caracter în set ^, care spune funcției **scanf()** să accepte caracterele care *nu* sunt definite în scanset.

Pentru a specifica o înșiruire folosiți o cratimă. De exemplu, pentru a spune lui **scanf()** să accepte caracterele de la A la Z scrieți:

```
%[A-Z]
```

Un lucru important de reținut este că scanset face diferențe între literele mari și cele mici. Dacă veți căuta aceleași litere, mari și mici, va trebui să le specificați separat.

Eliminarea spațiilor libere nedorite

Un caracter de spațiu liber într-un șir de control determină **scanf()** să sară peste unul sau mai multe caractere de acest tip din streamul de intrare. Un caracter de spațiu liber este un spațiu simplu, un spațiu de tabulare orizontală sau verticală, o pagină nouă sau un caracter de linie nouă. În esență, un caracter de spațiu liber într-un șir de control face ca **scanf()** să citească, dar nu să memoreze, orice număr (inclusiv zero) de caractere de spațiu liber, până la primul caracter care nu este de acest tip.

Caractere care nu sunt de tip spațiu liber în șirul de control

Un caracter care nu este de tip spațiu liber în șirul de control face ca **scanf()** să citească și să elimine din streamul de intrare caracterele similare lui. De exemplu, **"%d,%d"** determină ca **scanf()** să citească un întreg, să citească și să elimine o virgulă și să citească un întreg. Dacă acel caracter căutat nu este întâlnit, **scanf()** se va încheia. Dacă doriți să citiți și să eliminați un semn procent, folosiți în șirul de control **%%**.

Trebuie să transmiteți adrese în scanf()

Toate variabilele folosite pentru a primi valori prin **scanf()** trebuie să fie transmise prin adresele lor. Aceasta înseamnă că toate argumentele trebuie să fie pointeri la variabilele care primesc efectiv intrările. Amintiți-vă că acesta este modul lui C de a crea o apelare prin referință și el permite unei funcții să modifice conținutul unui argument. De exemplu, pentru a citi un întreg într-o variabilă **numara**, veți folosi următoarea apelare a funcției **scanf()**:

```
scanf("%d", &numara);
```

Șirurile vor fi citite în matrice de caractere iar numele matricei, fără indice, este adresa primului element al matricei. Astfel, pentru a citi un șir din matricea de caractere **sir**, veți putea folosi:

```
scanf("%s", sir);
```

În acest caz, **sir** este deja un pointer și nu trebuie să fie precedat de operatorul &.

Modelatori de format

Ca și **printf()**, **scanf()** permite ca un număr din specificatorii săi de format să fie modificați.

Specificatorii de format pot să includă modelatorul de lungime maximă a câmpului. Acesta este un întreg plasat între % și specificatorul de format, care limitează numărul de caractere citite din acel câmp. De exemplu, pentru a nu citi mai mult de 20 de caractere din **sir**, veți scrie:

```
scanf("%20s", sir);
```

Dacă streamul de intrare este mai mare de 20 de caractere, o apelare ulterioară a intrării va începe unde s-a încheiat prima. De exemplu, dacă introduceți:

```
ABCDEFGHIJKLMNOPQRSTUVWXYZ
```

ca răspuns al apelării lui **scanf()** pentru exemplul acesta, vor fi plasate în **sir** doar primele 20 de caractere, adică până la „T", datorită specificatorului de lungime maximă a câmpului. Aceasta înseamnă că UVWXYZ, caracterele rămase, nu au fost încă folosite. Dacă **scanf()** mai este apelată o dată astfel,

```
scanf("%s", sir);
```

în **str** vor fi plasate literele UVWXYZ. Intrarea dintr-un câmp se poate termina înainte ca să se ajungă la mărimea maximă a câmpului dacă se întâlneşte un spaţiu liber. În acest caz, **scanf()** se deplasează la următorul câmp.

Pentru a citi un întreg lung, puneţi un **l** (litera l mică) în faţa specificatorului de format. Pentru a citi un întreg scurt, puneţi un **h** în faţa specificatorului de format. Aceşti modelatori pot fi folosiţi împreună cu codurile de format **d**, **i**, **o**, **u** şi **x**.

Specificatorii **f**, **e** şi **g** spun implicit funcţiei **scanf()** să atribuie datele unei variabile **float**. Dacă puneţi un **l** (litera l mică) în faţa unuia dintre aceşti specificatori, **scanf()** atribuie datele unui **double**. Utilizarea unui **L** spune lui **scanf()** că variabila care primeşte datele este un **long double**.

Suprimarea intrărilor

Puteţi să îi spuneţi funcţiei **scanf()** să citească un câmp, dar să nu îl atribuie nici unei variabile precedând acel cod de format pentru câmp cu un *. De exemplu, dându-se

```
scanf("%d%*c%d", &x, &y);
```

puteţi să introduceţi perechea de coordonate **10,10**. Virgula va fi citită corect, dar nu va fi atribuită nici unei variabile. Suprimarea atribuirilor este utilă în special când doriţi să prelucraţi doar o parte din ceea ce a fost introdus.

Capitolul 9

I/O cu fişiere



După cum s-a menţionat în **Capitolul 8**, limbajul C nu conţine nici o instrucţiune de I/O. În locul lor, toate operaţiile de I/O au loc prin apelări ale funcţiilor din biblioteca standard C. Această caracteristică face sistemul de fişiere din C foarte puternic şi flexibil. Sistemul de I/O pentru C permite de asemenea, datelor să fie transferate ori în reprezentare internă binară, ori în format de text lizibil de către om, ceea ce permite cu uşurinţă crearea de fişiere care să corespundă oricăror necesităţi.

I/O pentru ANSI C faţă de I/O pentru Unix

Standardul ANSI C defineşte un set complet de funcţii de I/O care pot fi folosite pentru a citi şi scrie orice tip de date, spre deosebire de vechiul standard Unix C care conţine două sisteme distincte pentru fişiere care tratează operaţiile de I/O. Prima metodă este vag asemănătoare cu cea definită de standardul ANSI C şi mai este denumită sistemul de fişiere tip *buffer* (uneori este folosit în locul acestuia termenul *formatat* sau *de nivel înalt*). Cel de-al doilea este sistemul de fişiere *propriu pentru Unix* (denumit uneori *neformatat* sau *unbuffered*) şi este definit doar de standardul vechi pentru Unix. Standardul ANSI C nu defineşte sistemul de fişiere propriu pentru Unix deoarece, printre altele, cele două sisteme sunt redundante, iar sistemul de tip Unix nu poate fi aplicat în toate mediile care acceptă C.

Din motive asemănătoare, sistemul de fişiere propriu pentru Unix nu este definit sau acceptat de standardul ANSI propus pentru C++. Însă *toate compilatoarele de C++ acceptă sistemul de fişiere ANSI C*. Deoarece sistemul de fişiere din Unix nu este prea important pentru programarea în C++ şi nu este definit nici de standardul ANSI C şi nici de cel propus pentru ANSI C++, el nu este discutat în această carte.

I/O în C faţă de I/O în C++

Deoarece C constituie baza limbajului C++, există uneori întrebarea cum se corelează sistemul de fişiere din C cu cel din C++. Următoarea scurtă discuţie răspunde acestei întrebări.

C++ admite întreg sistemul de fişiere din ANSI C. De aceea, dacă veţi transporta pe viitor vechiul cod de C în C++ nu va trebui să schimbaţi toate rutinele de I/O. Dar, C++ defineşte şi propriul său sistem de I/O orientat pe obiecte, care include atât funcţii cât şi operatori de I/O. Sistemul C++ de I/O dublează complet funcţionalitatea sistemului I/O din C. În general, dacă veţi folosi C++ pentru a scrie programe orientate pe obiecte, veţi prefera să utilizaţi sistemul de I/O orientat pe obiecte. Altfel sunteţi liber să folosiţi atât sistemul de fişiere orientat pe obiecte, cât şi sistemul de fişiere din C. (Totuşi, majoritatea programatorilor preferă să folosească sistemul de I/O din C++ din motive care vor deveni clare în **Partea a doua** a acestei cărţi.)

Streamuri şi fişiere

Înainte de a începe discuţia despre sistemul de fişiere ANSI C, este important să înţelegeţi diferenţa între termenii *streamuri* şi *fişiere*. Sistemul de I/O din C asigură o interfaţă cu programatorul, coerenţă şi independenţă de aparatul folosit efectiv. Aceasta înseamnă că sistemul de I/O separă printr-o anumită abstractizare programatorul de echipament. Forma abstractă se numeşte *stream*, iar instrumentul efectiv, *fişier*. Este important să înţelegeţi cum interacţionează streamurile şi fişierele.



NOTĂ: Conceptele de stream şi de fişier sunt la fel de importante pentru sistemul de I/O discutat în Partea a doua a acestei cărţi.

Streamuri

Sistemul de fişiere din C este proiectat să lucreze cu o mare varietate de echipamente, care include terminale, drivere de disc şi drivere de unitate de bandă. Chiar dacă echipamentele diferă, sistemul de fişiere din C le transformă pe fiecare într-un instrument logic numit stream. Toate streamurile se comportă la fel. Deoarece streamurile sunt independente de echipamente, aceeaşi funcţie care poate să scrie într-un fişier de pe disc poate fi folosită, de asemenea, pentru a scrie la alt tip de dispozitiv, ca de exemplu o consolă. Există două tipuri de streamuri: text şi binar.

Streamuri de tip text

Un *stream de tip text* este o secvenţă de caractere. Standardul ANSI C permite (dar nu impune) ca un stream de tip text să fie organizat în linii terminate cu un caracter de linie nouă. Totuşi, caracterul de linie nouă din ultima linie este opţional, iar utilizarea sa este determinată de modul de implementare. (De fapt, majoritatea compilatoarelor de C/C++ nu încheie streamurile tip text cu un astfel de caracter.) Într-un stream pot să apară anumite transformări cerute de mediul gazdă. De exemplu, un caracter de linie nouă poate să fie convertit într-o pereche început de rând / linie nouă. De aceea, nu va exista o relaţie biunivocă între caracterele care sunt scrise (sau citite) şi cele de la echipamentul extern. De asemenea, datorită posibilelor modificări, numărul de caractere scrise (sau citite) poate să nu fie acelaşi cu cel din acele echipamente.

Streamuri binare

Un *stream binar* este o secvenţă de octeţi într-o corespondenţă biunivocă cu cei de la echipamentul extern - cu alte cuvinte, nu apar transformări de caractere. De

asemenea, numărul de octeţi scris (sau citit) este acelaşi cu numărul de la echipamentul extern. Totuşi, unui stream liniar i se poate adăuga un anumit număr de octeţi cu valoare nulă. Aceştia pot fi folosiţi pentru a completa informaţiile astfel încât să umple, de exemplu, un sector de pe un disc.

Fişiere

În C, un *fişier* poate să fie orice, de la un fişier de pe disc până la un terminal sau o imprimantă. Veţi asocia un stream cu un anumit fişier efectuând o operaţie de deschidere. O dată deschis un fişier, se poate efectua un schimb de informaţii între el şi programul dvs.

Nu toate fişierele au aceleaşi posibilităţi. De exemplu, un fişier de pe disc poate să admită un acces aleatoriu, în timp ce portul unui modem nu o poate face. Aceasta introduce ceva important pentru sistemul de I/O din C: toate streamurile sunt la fel, dar fişierele nu.

Dacă fişierul poate să admită *position requests (cereri de poziţionări)*, deschiderea acelui fişier iniţializează, de asemenea, *indicatorul de poziţie* către începutul fişierului. Pe măsură ce fiecare caracter este citit din sau scris în fişier, indicatorul de poziţie este incrementat, asigurând parcurgerea fişierului.

Un fişier se disociază de un anumit stream printr-o operaţie de închidere. Dacă închideţi un fişier deschis pentru ieşire, conţinutul (dacă există unul) streamului său asociat este scris la dispozitivul extern. Acest proces este în general numit *flushing (golire)* a streamului şi garantează că nici o informaţie nu va rămâne accidental în bufferul de pe disc. Toate fişierele se închid automat când programul se termină normal, fie prin întoarcerea funcţiei **main()** în sistemul de operare, fie printr-o apelare a funcţiei **exit()**. Când un program nu se termină normal, dacă se blochează sau dacă apelează **abort()**, fişierele nu se închid.

Fiecare stream care este asociat unui fişier are o structură de control pentru fişiere de tip **FILE**, definită în fişierul antet STDIO.H. Nu modificaţi niciodată acest bloc de control.

Dacă sunteţi proaspăt programator, distincţia pe care o face C între streamuri şi fişiere poate să vă pară inutilă sau forţată. Amintiţi-vă doar că scopul său principal este de a oferi o interfaţă coerentă. În C, pentru a realiza toate operaţiile de I/O, trebuie să gândiţi în termeni de stream şi să folosiţi doar un singur sistem de fişiere. Sistemul de I/O din C converteşte automat intrările sau ieşirile brute de la fiecare echipament într-un stream uşor de manevrat.

Bazele sistemului de fişiere

Sistemul de fişiere din ANSI C se compune din mai multe funcţii înrudite. Cele mai uzuale sunt prezentate în Figura 9-1. Aceste funcţii cer ca fişierul antet STDIO.H să fie inclus în orice program în care sunt folosite. Reţineţi că majoritatea lor încep cu litera „f", o reminiscenţă a standardului pentru Unix C, care defineşte două sisteme pentru fişiere. Funcţiile de I/O din Unix nu aveau prefix iar majoritatea funcţiilor sistemului de I/O formatată aveau prefixul „f". Comitetul de standardizare pentru C a preferat să menţină această convenţie asupra numelor în scopul continuităţii.

Fişierul antet STDIO.H asigură prototipurile pentru funcţiile de I/O şi defineşte următoarele tipuri: **size_t**, **fpos_t** şi **FILE**. Tipul **size_t** este o varietate de întreg fără semn, ca şi **fpos_t**. Tipul **FILE** va fi discutat în secţiunea următoare.

STDIO.H defineşte şi mai multe funcţii macro. Cele importante pentru acest capitol sunt **NULL**, **EOF**, **FOPEN_MAX**, **SEEK_SET**, **SEEK_CUR** şi **SEEK_END**. Funcţia macro **NULL** defineşte un pointer null. Funcţia macro **EOF** este, în general, definită ca -1 şi este valoarea returnată când o funcţie de intrare încearcă să citească peste sfârşitul fişierului. **FOPEN_MAX** defineşte o valoare întreagă ce determină numărul de fişiere care pot fi deschise în acelaşi timp. Celelalte sunt folosite cu **fseek()**, care este funcţia ce efectuează accesul aleatoriu într-un fişier.

Pointerul fişierului

Pointerul pentru fişier este legătura dintre fişier şi sistemul de I/O ANSI C. Un *pointer pentru fişier* este un pointer către informaţiile care definesc diferite lucruri despre fişier, cum ar fi numele, starea şi poziţia curentă a fişierului. În principal, pointerul pentru fişier identifică un anumit fişier de pe disc şi este folosit de către

Nume	Scop
fopen()	Deschide un fişier
fclose()	Închide un fişier
putc()	Scrie un caracter într-un fişier
fputc()	La fel ca putc()
getc()	Citeşte un caracter dintr-un fişier
fgetc()	La fel ca getc()
fseek()	Caută un anumit octet într-un fişier
fprintf()	Este pentru un fişier ceea ce este printf() pentru consolă
fscanf()	Este pentru un fişier ceea ce este scanf() pentru consolă
feof()	Returnează adevărat dacă se ajunge la sfârşitul fişierului
ferror()	Returnează adevărat dacă a apărut o eroare
rewind()	Readuce indicatorul de poziţie al fişierului la început
remove()	Şterge un fişier
fflush()	Goleşte un fişier

Tabelul 9-1 *Cele mai uzuale funcţii ale sistemului de fişiere ANSI C*

streamul asociat pentru a conduce operaţiile funcţiilor de I/O. Un pointer de fişier este o variabilă pointer de tip **FILE**. Pentru a citi sau a scrie fişiere, programul trebuie să folosească pointeri pentru ele. Pentru a obţine o variabilă de tip pointer pentru fişier folosiţi o instrucţiune ca aceasta:

```
FILE *fp;
```

Deschiderea unui fişier

Funcţia **fopen()** deschide un stream pentru a fi folosit şi îl asociază unui fişier. Apoi returnează pointerul de fişier asociat acelui fişier. Deseori (şi pentru restul acestei prezentări) fişierul este unul de pe disc. Funcţia **fopen()** are următorul prototip:

FILE *fopen(const char *numefişier*, const char *mod*);

Aici *numefişier* este un pointer către un şir de caractere care creează un nume de fişier valid şi poate include o specificare de cale. Şirul spre care indică *mod* determină modul în care va fi deschis fişierul. **Tabelul 9-2** prezintă valorile legale pentru *mod*. Şirurile ca „r+b" pot fi reprezentate şi ca „rb+".

Cum am mai spus, funcţia **fopen()** returnează un pointer de fişier. Programul dvs. nu trebuie să modifice niciodată valoarea acestui pointer. Dacă apare o eroare la încercarea de deschidere a fişierului, **fopen()** returnează un pointer null.

Mod	Semnificaţie
r	Deschide un fişier tip text pentru a fi citit
w	Creează un fişier tip text pentru a fi scris
a	Adaugă într-un fişier tip text
rb	Deschide un fişier de tip binar pentru a fi citit
wb	Creează un fişier de tip binar pentru a fi scris
ab	Adaugă într-un fişier de tip binar
r+	Deschide un fişier tip text pentru a fi citit/scris
w+	Creează un fişier tip text pentru a fi citit/scris
a+	Adaugă în sau creează un fişier tip text pentru a fi citit/scris
r+b	Deschide un text în binar pentru a fi citit/scris
w+b	Creează un fişier de tip binar pentru a fi citit/scris
a+b	Adaugă sau creează un fişier de tip binar pentru a fi citit/scris

Tabelul 9-2. *Valorile modurilor permise*

După cum arată **Tabelul 9-2**, un fişier poate fi deschis în mod text sau în mod binar. Pentru majoritatea instalărilor, în modul text, secvenţele retur de car/trecere la linie nouă sunt transformate în caractere de linie nouă. La ieşire, are loc procesul reciproc: caracterele de linie nouă sunt transformate în retur de car / trecere la linie nouă. Asemenea transformări nu au loc în fişierele de tip binar.

Următorul fragment foloseşte **fopen()** pentru a deschide fişierul numit TEST pentru ieşiri.

```
FILE *fp;
fp = fopen("test", "w");
```

Deşi tehnic este corect, veţi vedea de obicei acest cod scris astfel:

```
FILE *p;
if ((fp = fopen("test", "w"))==NULL) {
    printf("Nu pot deschide fisierul.\n");
    exit(1);
}
```

Această metodă va detecta orice eroare de deschidere a fişierului, cum ar fi o protecţie la scriere sau un disc plin, înainte ca programul să încerce să scrie. În general, este bine să aveţi confirmarea că **fopen()** a reuşit înainte de a încerca orice altă operaţie cu fişierul.

Dacă folosiţi **fopen()** pentru a deschide un fişier pentru scriere, orice fişier care există deja cu acel nume va fi şters şi va fi început un nou fişier. Dacă nu există fişiere cu acel nume, va fi creat unul. Dacă doriţi să adăugaţi la sfârşitul fişierului, trebuie să folosiţi modul „a". Puteţi deschide fişiere existente şi numai pentru operaţii de citire. Dacă fişierul nu există, va fi returnată o eroare. În sfârşit, dacă este deschis un fişier pentru operaţii de citire/scriere, el nu va fi şters dacă există. Dacă nu există, va fi creat.

Numărul de fişiere care pot fi deschise la un moment dat este specificat de **FOPEN_MAX**. Această valoare va fi de obicei mai mare de 8, dar trebuie să căutaţi în manualul compilatorului valoarea sa exactă.

Închiderea unui fişier

Funcţia **fclose()** închide un stream care a fost deschis prin apelarea funcţiei **fopen()**. Ea scrie în fişier orice dată rămasă în bufferul discului şi execută o închidere a fişierului la nivelul sistemului de operare. Eşecul închiderii unui stream implică tot felul de necazuri, inclusiv date pierdute, fişiere distruse şi posibile includeri de erori în programul dvs. De asemenea, **fclose()** eliberează blocul de control al fişierului asociat cu acel stream, făcându-l disponibil pentru a fi reutilizat. În majoritatea cazurilor există o limită a sistemului de operare relativ la numărul de

fişiere deschise la un moment dat, astfel încât va trebui să închideţi un fişier înainte de a deschide altul.

Funcţia **fclose()** are prototipul:

int fclose(FILE **fp*);

Aici *fp* este pointerul de fişier returnat de apelarea lui **fopen()**. O valoare de zero returnată reprezintă o operaţie de închidere realizată cu succes. Dacă apare vreo eroare funcţia returnează **EOF**. Puteţi folosi funcţia standard **ferror()** (discutată pe scurt) pentru a determina şi a semnala orice probleme. În general, **fclose()** va eşua doar când o dischetă a fost scoasă prea devreme din unitatea de disc sau când nu mai există spaţiu pe disc.

Scrierea unui caracter

Sistemul de I/O din ANSI C defineşte două funcţii echivalente care scriu caractere: **putc()** şi **fputc()**. (De fapt **putc()** este introdusă ca macro.) Au fost menţinute amândouă doar pentru a se păstra compatibilitatea cu versiunile vechi de C. Această carte foloseşte **putc()**, dar dacă doriţi puteţi folosi **fputc()**.

Funcţia **putc()** scrie caractere într-un fişier care a fost deschis anterior pentru a se scrie în el, folosindu-se funcţia **fopen()**. Prototipul acestei funcţii este :

int putc(int *ch*, FILE **fp*);

unde *fp* este pointerul de fişier returnat de **fopen()** iar *ch* este caracterul care va fi obţinut. Pointerul de fişier spune funcţiei **putc()** în care fişier de pe disc să scrie. Din motive istorice, *ch* este definit ca fiind **int**, dar este folosit doar octetul de ordin inferior.

Dacă o operaţie **putc()** se încheie cu bine, ea returnează caracterul scris. Altfel, ea returnează **EOF**.

Citirea unui caracter

Există de asemenea două funcţii echivalente care introduc un caracter: **getc()** şi **fgetc()**. Sunt definite ambele pentru a se păstra compatibilitatea cu versiunile de C vechi. Această carte foloseşte **getc()** (implementată de fapt ca macro), dar, dacă doriţi, puteţi să utilizaţi şi **fgetc()**.

Funcţia **getc()** citeşte caractere din fişiere deschise cu **fopen()** în modul de citire. Prototipul ei este :

int getc(FILE **fp*);

unde *fp* este pointerul de fişier de tip **FILE** returnat de **fopen()**. Din motive istorice **getc()** returnează un întreg, dar octetul de ordin superior este zero.

Funcţia **getc()** returnează **EOF** când s-a ajuns la sfârşitul fişierului. De aceea, pentru a citi până la sfârşitul unui fişier de tip text, puteţi folosi următorul cod:

```
do {
    ch = getc(fp);
}   while(ch!=EOF);
```

Dar **getc()** returnează **EOF** şi dacă apare o eroare. Pentru a determina exact ce s-a întâmplat puteţi folosi **ferror()**.

Utilizarea funcţiilor fopen(), getc(), putc() şi fclose()

Funcţiile **fopen()**, **getc()**, **putc()** şi **fclose()** constituie setul minim de rutine pentru fişiere. Următorul program, KTOD, este un exemplu simplu de utilizare a funcţiilor **putc()**, **fopen()** şi **fclose()**. El citeşte caracterele de la tastatură şi le scrie pe un fişier de pe disc până când utilizatorul tastează un semn pentru dolar. Numele fişierului este specificat în linia de comandă. De exemplu, dacă numiţi programul următor KTOD, comanda **KTOD TEST** vă permite să introduceţi linii de text în fişierul numit TEST.

```
/* KTOD: O cale spre programul de pe disc. */
#include <stdio.h>
#include <stdlib.h>

void main(int argc, char *argv[])
{
    File *fp;
    char ch;

    if(argc!=2) {
          printf("Ati uitat sa introduceti numele
                   fisierului.\n");
          exit(1);
    }

    if(fp=fopen(argv[1], "w"))==NULL {
          printf("Nu pot sa deschid fisierul.\n");
          exit(1);
    }

    do {
          ch = getchar();
```

```
        putc(ch, fp);
    }   while (ch!= 'S');

    fclose(fp);
}
```

Programul complementar DTOS, prezentat mai jos, citeşte orice fişier ASCII şi îi afişează conţinutul pe ecran.

```
/* DTOS: Un program care citeste fisiere si le afiseaza pe
   ecran. */
#include <stdio.h>
#include <stdlib.h>

void main(int argc, char *argv[])
{
    FILE *fp;
    char ch;

    if(argc!=2) {
        printf("Ati uitat sa introduceti numele
                  fisierului.\n");
        exit(1);
    }

    if((fp=fopen(argv[1], "r"))==NULL) {
        printf("Nu pot sa deschid fisierul.\n");
        exit(1);
    }
    ch = getc(fp);          /* citeste un caracter */

    while (ch!=EOF) {
        putchar(ch);        /* afiseaza pe ecran */
        ch = getc(fp);
    }

    fclose(fp);
}
```

Încercaţi aceste două programe. Mai întâi rulaţi KTOD pentru a crea un fişier de tip text. Apoi citiţi-i conţinutul folosind DTOS.

Utilizarea funcţiei feof()

După cum am spus mai devreme, sistemul de fişiere ANSI C poate, de asemenea, să opereze cu date binare. Când se deschide un fişier pentru intrări binare, poate fi citită o valoare întreagă egală cu **EOF**. Aceasta va determina ca rutina de intrare să indice o condiţie de sfârşit de fişier chiar dacă nu s-a ajuns la sfârşitul fizic al fişierului. Pentru a rezolva această problemă, C include funcţia **feof()**, care stabileşte când a fost atins sfârşitul acelui fişier. Funcţia **feof()** are următorul prototip:

int feof(FILE *fp);

Ca şi celelalte funcţii pentru fişiere, prototipul său se află în STDIO.H. **feof()** returnează adevărat dacă s-a ajuns la sfârşitul fişierului; altfel returnează 0. Astfel, următoarea rutină citeşte un fişier în binar până când se întâlneşte sfârşitul fişierului.

```
while(!feof(fp)) ch = getc(fp);
```

Desigur, această metodă o puteţi aplica şi fişierelor tip text, nu doar celor binare.

Următorul program, care copiază fişiere tip text sau binar, conţine un exemplu pentru **feof()**. Fişierele sunt deschise în mod binar, iar **feof()** caută sfârşitul fişierului.

```
/* Copiaza un fisier. */
#include <stdio.h>
#include <stdlib.h>

void main(int argc, char *argv[])
{
    FILE *intra, *iese;
    char ch;

    if(argc!=3) {
        printf("Ati uitat sa introduceti un nume de
                  fisier.\n");
        exit(1);
    }

    if((intra=fopen(argv[1], "rb"))==NULL) {
        printf("Nu pot sa deschid fisierul sursa.\n");
        exit(1);
```

```
    }
    if((iese=fopen(argv[2], "wb"))==NULL) {
          printf("Nu pot sa deschid fisierul
                   destinatie.\n");
          exit(1);

    /* Acest cod copiaza efectiv fisierul. */
    while (!feof(intra)) {
          ch = getc(intra);
          if(!feof(intra)) putc(ch, iese);
    }

    fclose(intra);
    fclose(iese);
}
```

Lucrul cu şirurile: fputs() şi fgets()

În plus faţă de **getc()** şi **putc()**, C admite funcţiile înrudite **fputs()** şi **fgets()**, care citesc şi scriu şiruri de tip caracter din sau într-un fişier de pe disc. Aceste funcţii lucrează exact la fel ca **putc()** şi **getc()**, dar în loc să citească sau să scrie un singur caracter, citesc sau scriu şiruri. Ele au următoarele prototipuri:

int fputs(const char **sir*, FILE **fp*);
char *fgets(char **sir*, int *lungime*, FILE **fp*);

Prototipurile pentru **fputs()** şi **fgets()** se află în STDIO.H.

Funcţia **fputs()** scrie în streamul specificat şirul spre care indică *sir*. Dacă apare o eroare ea returnează **EOF**.

Funcţia **fgets()** citeşte un şir din streamul specificat până când este întâlnit un caracter de linie nouă sau au fost citite *lungime*-1 caractere. Dacă este citit un caracter de linie nouă, el este inclus în şir (spre deosebire de cazul funcţiei **gets()**). Şirul rezultat va fi terminat cu null. Funcţia returnează *sir* dacă reuşeşte şi un pointer dacă apare o eroare.

Următorul program exemplifică **fputs()**. Citeşte şiruri de la tastatură şi le scrie în fişierul cu numele TEST. Pentru a încheia programul, introduceţi o linie liberă. Deoarece **gets()** nu memorează caracterul de linie nouă, se adaugă unul înainte ca fiecare şir să fie scris în fişier, astfel încât fişierul să fie citit mai uşor.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
```



```
void main(void)
{
    char sir[80];
    FILE *fp;

    if((fp = fopen("TEST", "w"))==NULL) {
          printf("Nu pot deschide fisierul.\n");
          exit(1);
    }

    do {
          printf("Introduceti un sir (ENTER pentru a iesi
                   din program):\n");
          gets(sir);
          strcat(sir, "\n"); /* adauga trecerea la o linie
                                 noua */
          fputs(sir, fp);
    }     while(*sir!= '\n');
}
```

rewind()

Funcţia **rewind()** readuce indicatorul de poziţie al fişierului la început, indicatorul fiind specificat ca un argument. Aceasta înseamnă că ea „rederulează" fişierul. Prototipul ei este:

void rewind(FILE **fp*);

unde *fp* este un pointer valid pentru fişier. Prototipul funcţiei **rewind()** se află în STDIO.H.

Ca să vedeţi un exemplu de **rewind()** puteţi modifica programul din paragraful anterior, astfel încât să afişeze conţinutul fişierului abia creat. Pentru a realiza aceasta, programul rederulează fişierul după ce intrarea este completă şi apoi foloseşte **fgets()** pentru a-l citi. Reţineţi că fişierul trebuie să fie deschis acum în modul citire/scriere, folosindu-se „w+" ca parametru de mod.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

void main(void)
```

```
{
    char sir[80];
    FILE *fp;

    if((fp = fopen("TEST", "w"))==NULL) {
          printf("Nu pot deschide fisierul.\n");
          exit(1);
    }

    do {
          printf("Introduceti un sir (ENTER pentru a iesi
                    din program):\n");
          gets(sir);
          strcat(sir, "\n"); /* adauga trecerea la o linie
                                   noua */
          fputs(sir, fp);
    }     while(*sir!= '\n');
    /* acum citeste si afiseaza fisierul */
    rewind(fp); /* readuce indicatorul de pozitie al
                   fisierului la inceputul fisierului. */
    while(!feof(fp)) {
          fgets(sir, 79, fp);
          printf(sir);
    }
}
```

ferror()

Funcția **ferror()** determină dacă o operație cu fișiere a produs o eroare. Ea are prototipul:

int ferror(FILE *fp*);

Aici, *fp* este un pointer valid pentru fișier. Ea returnează adevărat dacă a apărut o eroare în timpul ultimei operații cu fișierul; altfel, returnează fals. Deoarece orice operație cu fișiere activează condiția de eroare, **ferror()** trebuie să fie apelată imediat după fiecare operație cu fișiere; altfel, se poate pierde o eroare. Prototipul pentru **ferror()** se află în STDIO.H.

Următorul program ilustrează **ferror()** înlăturând spații de tabulare dintr-un fișier de tip text și înlocuindu-le cu numărul corespunzător de spații simple. Mărimea spațiului de tabulare este definită de **TAB_SIZE**. Rețineți că **ferror()** este apelată după fiecare operație pe disc. Pentru a folosi programul specificați în linia de comandă numele fișierelor de intrare și de ieșire.


```
/* Programul inlocuieste spatiul de tabulare dintr-un
   fisier de tip text si verifica erorile. */
#include <stdio.h>
#include <stdlib.h>

#define TAB_SIZE 8
#define IN 0
#define OUT 1

void err(int e)

void main(int argc, char *argv[])
{
    FILE *in, *out;
    int tab, i;
    char ch;

    if(argc!=3) {
          printf("utilizare: lungimetab <intrare>
                     <iesire>\n");
          exit(1);
    }

    if((in = fopen(argv[1], "rb"))==NULL) {
          printf("Nu pot sa deschid %s.\n", argv[1]);
          exit(1);
    }

    if((out = fopen(argv[2], "wb"))==NULL) {
          printf("Nu pot sa deschid %s.\n", argv[2]);
          exit(1);
    }

    tab = 0;
    do {
          ch = getc(in);
          if(ferror(in)) err(IN);

          /* daca este gasit un tab, se scrie numarul
             corespunzator de spatii */
          if(ch=='\t') {
```

```
                for(i=tab; i<8; i++) {
                   putc(' ', out);
                   if(ferror(out)) err(OUT);
                }
                tab = 0
            }
            else {
                putc(ch, out);
                if(ferror(out)) err(OUT);
                tab++;
                if(tab==TAB_SIZE) tab = 0;
                if(ch=='\n' || ch=='\r') tab = 0;
            }
    } while(!feof(in));
    fclose(in);
    fclose(out);
}

void err(int e)
{
    if(e==IN) printf("Eroare la intrare. \n");
    else printf("Eroare la iesire. \n");
    exit(1);
}
```

Ştergerea fişierelor

Funcţia **remove()** şterge fişierul specificat. Prototipul ei este:

int remove(const char *numefişier*);

Ea returnează zero dacă se încheie cu succes. Altfel, returnează o valoare diferită de zero.

Următorul program şterge fişierul specificat în linia de comandă. Totuşi, ea vă mai oferă o şansă să vă răzgândiţi. O astfel de utilitate poate fi de folos începătorilor în lucrul cu calculatorul.

```
/* Verifica de doua ori inainte de a sterge. */
#include <stdio.h>
#include <stdlib.h>
#include <ctype.h>
```



```
main(int argc, char *argv[])
{
    char sir[80];

    if(argc!=2) {
        printf("Utilizare: xsterge <numefisier>\n");
        exit(1);
    }

    printf("Sterg %s? (Y/N): ", argv[1]);
    gets(sir);

    if(toupper(*sir)== 'Y')
        if(remove(argv[1])) {
            printf("Nu pot sa sterg fisierul.\n");
            exit(1);
        }
    return 0;  /* intoarcere cu succes in OS */
}
```

Golirea unui stream

Dacă doriţi să goliţi conţinutul unui stream de ieşire, folosiţi funcţia **fflush()**, al cărei prototip este prezentat aici:

int fflush(FILE *fp*);

Această funcţie scrie conţinutul datelor din buffer în fişierul asociat cu *fp*. Dacă apelaţi **fflush()** cu un *fp* nul, vor fi golite toate fişierele deschise pentru ieşiri.

Funcţia **fflush()** returnează 0 dacă a reuşit; altfel, returnează **EOF**.

fread() şi fwrite()

Pentru a citi şi a scrie tipuri de date care sunt mai mari de un octet, sistemul pentru fişiere din ANSI C asigură două funcţii: **fread()** şi **fwrite()**. Ele permit citirea şi scrierea blocurilor de orice tip de date. Prototipurile lor sunt:

size_t fread(void *buffer*, size_t *numar_octeţi*, size_t *numara*, FILE *fp*);
size_t fwrite(const void *buffer*, size_t *numar_octeti*, size_t *numara*, FILE *fp*);

Pentru **fread()** *buffer* este un pointer către o regiune de memorie care va primi datele de la fişier. Pentru **fwrite()**, *buffer* este un pointer către informaţiile care vor

fi scrise în acel fişier. Valoarea *numara* determină numărul de elemente citite sau scrise, fiecare având un număr de octeţi egal cu *numar_octeti*. (Amintiţi-vă că tipul **size_t** este definit în STDIO.H şi este aproximativ echivalent cu un întreg fără semn.) În sfârşit, *fp* este un pointer pentru fişier către un stream deschis anterior. Prototipurile pentru ambele funcţii sunt definite în STDIO.H.

Funcţia **fread()** returnează numărul de elemente citite. Această valoare poate fi mai mică decât *numara* dacă se ajunge la sfârşitul fişierului sau dacă apare o eroare. Funcţia **fwrite()** returnează numărul de elemente scrise. Această valoare va fi egală cu *numara*, dacă nu apare o eroare.

Utilizarea lui fread() şi fwrite()

Atât timp cât fişierul a fost deschis pentru date în mod binar, **fread()** şi **fwrite()** pot să scrie şi să citească orice tip de informaţii. De exemplu, următorul program scrie şi apoi citeşte un **double,** un **int** şi un **long** în şi dintr-un fişier de pe disc. Reţineţi cum foloseşte **sizeof** pentru a determina mărimea fiecărui tip de date.

```
/* Scrie unele date care nu sunt caractere intr-un fisier
   de pe disc si le citeste. */
#include <stdio.h>
#include <stdlib.h>

void main(void)
{
    FILE *fp;
    double d = 12.23;
    int i = 101;
    long l = 123023L;

    if((fp=fopen("test", "wb+"))==NULL) {
          printf("Nu pot sa deschid fisierul.\n");
          exit(1);
    }

    fwrite(&d, sizeof(double), 1, fp);
    fwrite(&i, sizeof(int), 1, fp);
    fwrite(&l, sizeof(long), 1, fp);

    rewind(fp);

    fread(&d, sizeof(double), 1, fp);
    fread(&i, sizeof(int), 1, fp);
    fread(&l, sizeof(long), 1, fp);
```



```
    printf("%f %d %ld", d, i, l);
    fclose(fp);
}
```

Aşa cum ilustrează acest program, bufferul poate fi (şi deseori este) chiar memoria folosită pentru a păstra valoarea. În acest program simplu, valorile returnate de **fread()** şi de **fwrite()** sunt ignorate. Totuşi, în practică, trebuie să verificaţi valoarea returnată de ele pentru a nu exista erori.

Una dintre cele mai uzuale aplicaţii pentru **fread()** şi **fwrite()** implică citirea şi scrierea tipurilor de date definite de utilizator, în special a structurilor. De exemplu, dându-se structura,

```
struct tip_struct {
    float bilant;
    char nume[80];
}   cust;
```

următoarea instrucţiune scrie conţinutul din **cust** în fişierul spre care indică **fp**.

```
fwrite(&cust, sizeof(struct tip_struct), 1, fp);
```

fseek() şi I/O în acces aleatoriu

Puteţi să efectuaţi operaţii de citire şi de scriere aleatorie folosind sistemul de I/O din ANSI C cu ajutorul funcţiei **fseek()**, care controlează indicatorul de poziţie al fişierului. Prototipul ei arată astfel:

int fseek(FILE **fp*, long *numocteti*, int *origine*);

Aici, *fp* este un pointer pentru fişier returnat de o apelare a funcţiei **fopen()**. *numocteti* este numărul de octeţi de la *origine* care va deveni noua poziţie curentă, iar *origine* este una dintre următoarele definiţii macro din STDIO.H.

Origine	**Nume macro**
Începutul fişierului	SEEK_SET
Poziţie curentă	SEEK_CUR
Sfârşit de fişier	SEEK_END

Astfel, pentru a căuta *numocteti* de la începutul fişierului, *origine* va fi **SEEK_SET**. Pentru a porni de la poziţia curentă, folosiţi **SEEK_CUR** iar pentru a căuta de la sfârşitul fişierului folosiţi **SEEK_END**. Funcţia **fseek()** returnează 0

când se încheie cu succes şi o valoare diferită de zero când apare o eroare.

Următorul fragment ilustrează **fseek()**. Ea caută şi afişează octetul specificat din fişierul specificat. Specificaţi pe linia de comandă mai întâi numele fişierului şi apoi octetul pe care îl căutaţi.

```
#include <stdio.h>
#include <stdlib.h>

void main(int argc, char *argv[])

{
    FILE *fp;

    if(argc!=3) {
          printf("Utilizare: SEEK numefisier octet\n");
          exit(1);
    }

    if((fp = fopen(argv[1], "r"))==NULL {
          printf("Nu pot sa deschid fisierul. \n");
          exit(1);
    }

    if(fseek(fp, atol(argv[2]), SEEK_SET)) {
          printf("Eroare la cautare.\n");
          exit(1);
    }

    printf("Octetul de la %ld este %c.\n", atol(argv[2]),
             getc(fp));
    fclose(fp);
}
```

Puteţi să folosiţi **fseek()** pentru a căuta multipli de orice tip de date prin simpla înmulţire a mărimii datei cu numărul de elemente pe care îl doriţi. De exemplu, să presupunem că aveţi o listă pentru poştă care constă din structuri de tip **tip_lista**. Pentru a căuta cea de a zecea adresă din fişierul care conţine adresele, folosiţi această instrucţiune:

```
fseek(fp, 9*sizeof(struct tip_lista), SEEK_SET);
```


fprintf() şi fscanf()

În plus faţă de funcţiile de bază pentru I/O deja discutate, sistemul ANSI C include şi **fprintf()** şi **fscanf()**. Aceste funcţii se comportă exact ca şi **printf()** şi **scanf()**, doar că lucrează cu fişiere. Prototipurile pentru **fprintf()** şi **fscanf()** sunt:

int fprintf(FILE **fp*, const char **şir_control*,...);
int fscanf(FILE **fp*, const char **şir_control*,...);

unde *fp* este un pointer de fişier returnat de o apelare a funcţiei **fopen()**. **fprintf()** şi **fscanf()** efectuează operaţiile asupra fişierului spre care indică *fp*.

Ca un exemplu, următorul program citeşte un şir şi un întreg de la tastatură şi le scrie într-un fişier de pe disc numit TEST. Apoi programul citeşte fişierul şi afişează informaţia pe ecran. După rularea acestui program, verifică fişierul TEST. După cum veţi vedea, el conţine un text lizibil de către om.

```
/* exemple de fscanf() - fprintf() */
#include <stdio.h>
#include <io.h>
#include <stdlib.h>

void main(void)
{
    FILE *fp;
    char s[80];
    int t;

    if((fp=fopen("test", "w")) == NULL) {
          printf("Nu pot sa deschid fisierul.\n");
          exit(1);
    }

    printf("Introduceti un sir si un numar: ");
    fscanf(stdin, "%s%d", s, &t); /* citeste de la
                                        tastatura */
    fprintf(fp, "%s %d", s, t); /* scrie in fisier */
    fclose(fp);

    if((fp=fopen("test", "r")) == NULL) {
          printf("Nu pot sa deschid fisierul. \n");
          exit(1);
    }
```

```
    fscanf(fp, "%s%d", s, &t); /* citeste din fisier */
    fprintf(stdout, "%s %d, s, t); /* afiseaza pe ecran */
}
```



*ATENŢIE: Chiar dacă **fprintf()** şi **fscanf()** sunt de multe ori cea mai simplă cale de a scrie şi de a citi date amestecate în sau din fişiere de pe disc, ele nu sunt mereu cele mai eficiente. Deoarece datele în format ASCII sunt scrise exact aşa cum s-ar afişa pe ecran (şi nu în binar), cele în plus sunt reluate la fiecare apelare. Astfel că, dacă viteza sau mărimea fişierului sunt importante, veţi folosi probabil **fread()** şi **fwrite()**.*

Streamurile standard

Întotdeauna când un program în C îşi începe execuţia, se deschid automat trei streamuri. Ele sunt **stdin** (intrare standard), **stdout** (ieşire standard) şi **stderr** (eroare standard). Normal, aceste streamuri se referă la consolă, dar ele pot fi redirecţionate de către sistemul de operare către elemente ale mediului care admit redirecţionarea I/O. (Redirecţionarea I/O este admisă de exemplu de Windows, DOS, Unix şi OS/2.)

Deoarece streamurile standard sunt pointeri de fişiere, ele pot fi folosite de către sistemul de fişiere ANSI C pentru a efectua operaţii de I/O pentru consolă. De exemplu, **putchar()** poate fi definită astfel:

```
putchar(char c)
{
    return putc(c, stdout);
}
```

În general, **stdin** este folosit pentru a citi de la consolă iar **stdout** şi **stderr** pentru a scrie la consolă. Puteţi să utilizaţi **stdin, stdout** şi **stderr** ca pointeri de fişier în orice funcţie care foloseşte o variabilă de tipul **FILE***. De exemplu, puteţi folosi **fputs()** pentru a scrie un şir la consolă folosind o astfel de apelare:

```
fputs ("va salut", stdout);
```

Reţineţi că **stdin, stdout** şi **stderr** nu sunt variabile în adevăratul sens al cuvântului şi nu li se pot atribui valori folosind **fopen()**. De asemenea, deoarece aceşti pointeri de fişiere sunt creaţi automat la începutul programului, ei sunt închişi automat la sfârşit; nu trebuie să încercaţi să-i închideţi.

Conectarea I/O la consolă

Amintiţi-vă din Capitolul 8 că pentru C practic nu există o deosebire între I/O pentru consolă şi pentru fişiere. Funcţiile de I/O la consolă descrise în Capitolul 8 direcţionează operaţiile lor către **stdin** sau către **stdout**. În esenţă, funcţiile de la consolă sunt doar versiuni speciale ale funcţiilor corespunzătoare pentru fişiere. Motivul existenţei lor este o facilitate pentru dvs., programatorii.

După cum s-a menţionat în paragraful anterior, puteţi să efectuaţi I/O la consolă folosind oricare dintre funcţiile sistemului de fişiere din C. Totuşi, ceea ce poate vă va surprinde va fi faptul că puteţi să efectuaţi I/O pentru fişiere de pe disc folosind funcţiile de I/O pentru consolă, ca de exemplu **printf()**! Aceasta deoarece toate funcţiile de I/O pentru consolă descrise în **Capitolul 8** operează asupra streamurilor **stdin** şi **stdout**. În mediile care permit redirecţionarea I/O, înseamnă că **stdin** şi **stdout** pot să se refere la alte elemente, altele decât tastatura şi ecranul. De exemplu, să luăm programul:

```
#include <stdio.h>

void main(void)
{
    char sir[80];

    printf("Introduceti un sir: ");
    gets(sir);
    printf(sir);
}
```

Să presupunem că acest program se numeşte TEST. Dacă executaţi TEST normal, el afişează un mesaj pe ecran, citeşte un şir de la tastatură şi îl afişează pe ecran. Dar, într-un mediu care admite redirecţionarea I/O, atât **stdin** cât şi **stdout**, sau ambele, pot fi redirecţionate spre un fişier. De exemplu, în mediile DOS sau Windows, executând următoarea comandă:

```
TEST > IESIRE
```

se ajunge ca rezultatele programului TEST să fie scrise într-un fişier numit OUTPUT. Executarea programului TEST după cum urmează:

```
TEST < INTRARE > IESIRE
```

direcţionează **stdin** către fişierul numit INTRARE şi transmite ieşirea spre un fişier numit IESIRE.



REȚINEȚI: Când se termină un program în C, toate streamurile redirecționate sunt automat readuse la starea lor implicită.

Utilizarea funcției freopen() pentru redirecționarea streamurilor standard

Puteți să redirecționați streamurile standard folosind funcția **freopen()**. Aceasta asociază un stream existent cu un nou fișier. Astfel, puteți să o folosiți pentru a asocia un stream standard cu un nou fișier. Prototipul său este:

FILE *freopen(const char *numefișier*, const char *mod*, FILE *stream*);

unde *numefisier* este un pointer către numele fișierului pe care doriți să îl asociați cu streamul spre care indică *stream*. Fișierul se deschide după cum indică *mod*, care poate să aibă aceleași valori ca și cele pentru **fopen()**. **freopen** returnează *stream* dacă nu apare o eroare; altfel, returnează NULL.

Următorul program folosește **freopen()** pentru a redirecționa **stdout** spre un fișier numit IESIRE.

```
#include <stdio.h>

void main(void)
{
    char sir[80];

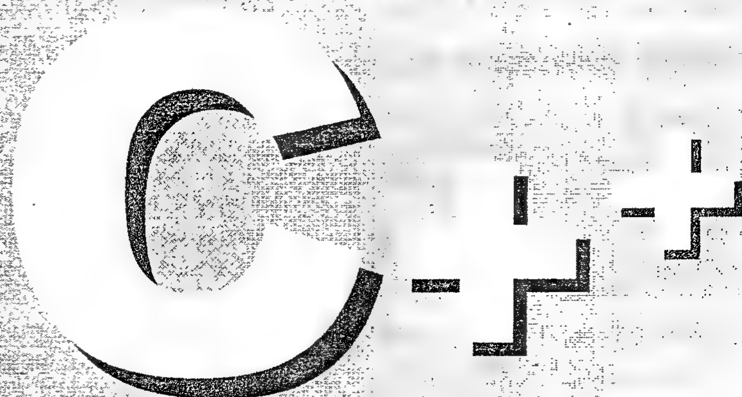
    freopen("IESIRE", "w", stdout);

    printf("Introduceti un sir: ");
    gets(sir);
    printf(sir);
}
```

În general, redirecționarea streamului standard utilizând **freopen()** este folositoare în situații speciale, așa cum este depanarea. Dar executarea operațiilor I/O pe disc cu ajutorul lui **stdin** și **stdout** nu este la fel de eficientă ca utilizarea funcțiilor de tip **fread()** sau **fwrite()**.

Capitolul 10

Preprocesorul. Comentarii



În codul sursă al unui program în C sau C++ puteți să includeți diverse instrucțiuni pentru compilator. Acestea se numesc *directive pentru preprocesor* și, deși nu fac parte din limbajul C sau C++, lărgesc sfera de acțiune a programelor în C/C++. Acest capitol studiază comentariile.

Preprocesorul

Preprocesorul conține următoarele directive:

#if	#include
#ifdef	#define
#ifndef	#undef
#else	#line
#elif	#error
#endif	#pragma

După cum puteți vedea, toate directivele preprocesorului încep cu semnul #. În plus, fiecare dintre ele trebuie să fie pe o linie separată. De exemplu,

```
#include <stdio.h>   #include <stdlib.h>
```

nu va funcționa.

#define

Directiva **#define** definește un identificator și o secvență (un set) care va înlocui identificatorul de fiecare dată când acesta este întâlnit în codul sursă. Identificatorul se numește *nume macro* iar procesul de înlocuire *înlocuire macro*. Forma generală a directivei este:

#define *nume_macro secventa_de_caractere*

Rețineți că această instrucțiune nu necesită punct și virgulă. Între identificator și secvența de caractere poate să existe orice număr de spații, dar odată începută secvența, ea se termină doar cu un sfârșit de linie.

De exemplu, dacă doriți să folosiți în program cuvântul **ADEVARAT** în locul valorii 1 și **FALS** pentru 0, puteți să declarați aceste două definiții macro astfel:

```
#define ADEVARAT 1
#define FALS 0
```

Aceasta determină compilatorul să înlocuiască cu 1 sau 0 toate aparițiile cuvintelor **ADEVARAT** sau **FALS** din codul sursă. De exemplu, următoarea instrucțiune afișează pe ecran **0 1 2**.

```
printf("% d% d %d", FALS, ADEVARAT, ADEVARAT+1);
```

O data ce a fost definit un macro, el poate fi folosit ca parte a definiției unui alt nume de macro. De exemplu, acest cod definește valorile lui **UNU**, **DOI** și **TREI**:

```
#define UNU          1
#define DOI          UNU+UNU
#define TREI         UNU+DOI
```

Macrosubstituția este simpla înlocuire a unui identificator cu secvența de caractere asociată. Astfel, dacă doriți să definiți un mesaj de eroare standard, puteți scrie ceva de genul acesta:

```
#define E_MS "eroare de intrare standard \n"
/*... */
printf(E_MS);
```

Când va întâlni identificatorul **E_MS**, compilatorul îl va înlocui cu șirul "eroare de intrare standard \n". Pentru compilator, instrucțiunea **printf()** va reprezenta efectiv:

```
printf("eroare de intrare standard \n");
```

Dacă un identificator este închis între ghilimele, nu se va efectua substituția textului. De exemplu,

```
#define XYZ acesta este un test
printf("XYZ");
```

nu va afișa "acesta este un test", ci "XYZ".

Dacă grupul de caractere este mai lung decât o linie, puteți să îl continuați pe următorul plasând un backslash la sfârșitul acesteia, așa cum se prezintă aici:

```
#define SIR_LUNG "acesta este un sir \
foarte lung care este folosit ca exemplu"
```

De obicei programatorii de C/C++ folosesc literele mari pentru a defini identificatorii. Această convenție ajută pe oricine citește programul să știe dintr-o privire că va avea loc o macrosubstituție. De asemenea, este bine să se pună **#define** la începutul fișierului sau într-un fișier antet separat, nu să le împrăștiem în întregul program.

Definițiile macro sunt cel mai des folosite pentru a defini nume pentru „numere magice" care apar într-un program. De exemplu, puteți să aveți un program care definește o matrice și are mai multe rutine cu acces la matrice. În locul „codificării hard" a mărimii matricei printr-o constantă, puteți defini mărimea folosind instrucțiunea **#define** și apoi folosi numele macro de câte ori este nevoie de mărimea matricei. În acest fel, dacă trebuie modificată mărimea matricei, nu aveți altceva de făcut decât să o schimbați în instrucțiunea **#define** și apoi să recompilați programul. De exemplu,

```
#define MARIME_MAX 100
/*... */
float bilant[MARIME_MAX];
/* ... */
for(i=0; i<MARIME_MAX; i++) printf("%f", bilant[i]);
```

Deoarece **MARIME_MAX** definește mărimea matricei **bilant**, dacă ea va trebui modificată în viitor, va trebui doar să schimbați definiția pentru **MARIME_MAX**. Toate referirile ulterioare la ea vor fi aduse la zi automat când recompilați programul.

Definirea funcțiilor macro

Directiva **#define** are o altă caracteristică puternică: numele de macro poate să aibă argumente. De fiecare dată când este întâlnit un nume de macro, argumentele folosite în definirea sa sunt înlocuite cu argumentele efective din program. Această formă de macro este numită *funcție macro*. De exemplu,

```
#include <stdio.h>

#define ABS(a)  (a)<0 ? -(a) : (a)

void main(void)
{
    printf("abs de -1 si 1: %d %d", ABS(-1), ABS(1));
}
```

La compilarea acestui program, a din definirea macro va fi înlocuit cu valorile -1 și 1. Parantezele care îl încadrează pe **a** asigură o înlocuire corectă în toate cazurile. De exemplu, dacă parantezele pentru **a** sunt înlăturate, expresia

```
ABS(10-20)
```

va fi înlocuită cu:


```
10-20<0 ? -10-20 : 10-20
```

și va determina un rezultat greșit.

Utilizarea unei funcții macro în locul uneia propriu-zise are un mare avantaj: mărește viteza de execuție a codului deoarece nu apar întârzierile produse de apelările funcției. Totuși, dacă mărimea funcției macro este foarte mare, creșterea vitezei poate fi plătită cu o mărire a programului datorată dublării codului.



***NOTĂ:** Deși funcțiile macro cu parametri sunt o caracteristică valoroasă, veți vedea în Partea a doua a acestei cărți că C++ are un mod mai bun de a crea cod inline care nu se bazează pe construcții macro.*

#error

Directiva **#error** forțează oprirea compilării. Ea este folosită în primul rând pentru depanare. Forma generală a acestei directive este:

#error *mesaj_eroare*

mesaj_eroare nu se află între ghilimele duble. Când este întâlnită directiva **#error** se afișează mesajul de eroare, împreună cu eventuale alte informații stabilite de proiectantul compilatorului.

#include

Directiva **#include** spune compilatorului să citească și un fișier sursă în afară de cel care conține această directivă. Numele fișierului sursă adițional trebuie închis între ghilimele duble sau paranteze unghiulare. De exemplu,

```
#include "stdio.h"
#include <stdio.h>
```

spun ambele compilatorului de C/C++ să citească și să compileze fișierul antet pentru sistemul de fișiere al funcțiilor de bibliotecă.

Fișierele incluse pot să conțină alte directive tip **#include**. Acestea se numesc *includeri imbricate*. Numărul permis de niveluri de imbricare diferă în funcție de compilatoare. Standardul ANSI C stipulează că trebuie să fie permise cel puțin opt niveluri de includeri imbricate. Standardul propus pentru ANSI C++ recomandă să fie admise cel puțin 256 de niveluri de imbricare.

Modul de închidere a numelui fișierului între ghilimele sau paranteze unghiulare determină cum se face căutarea fișierului specificat. Dacă el este închis între

paranteze unghiulare, fişierul este căutat într-un mod definit de către proiectantul compilatorului. Deseori aceasta înseamnă să caute în anumite directoare realizate special pentru fişierele incluse. Dacă numele fişierului este închis între ghilimele, fişierul este căutat în altă manieră stabilită de implementare. Pentru multe compilatoare, aceasta înseamnă să caute în directorul în care se lucrează curent. Dacă fişierul nu este găsit, căutarea se reia, de data aceasta ca şi cum fişierul ar fi fost încadrat între paranteze unghiulare.

Uzual, pentru a include fişierele antet standard, majoritatea programatorilor folosesc parantezele unghiulare. Ghilimelele sunt utilizate pntru fişierele antet legate strict de fişierul cu care se lucrează. Dar nu există o regulă care să ceară acest lucru.

Directivele de compilare condiţionată

Există mai multe directive care vă permit compilarea selectivă a unor porţiuni din codul sursă al programului. Acest proces este numit *compilare condiţionată* şi este utilizat mult de companiile de soft care asigură şi dezvoltă multe versiuni înrudite ale aceluiaşi program.

#if, #else şi #endif

Probabil că cele mai utilizate directive de compilare condiţionată sunt **#if**, **#else**, **#elif** şi **#endif**. Aceste directive vă permit să introduceţi condiţionat porţiuni de cod bazate pe rezultatul unei expresii constante.

Forma generală pentru **#if** este:

#if *expresie_constanta*
    secventa de instrucţiuni
#endif

Dacă expresia constantă care urmează cuvântului cheie **#if** este adevărată, codul care se află între el şi **#endif** este compilat. Altfel, acel cod este ignorat. Directiva **#endif** marchează sfârşitul unui bloc **#if**. De exemplu,

```
/* Un exemplu simplu de #if. */
#include <stdio.h>

#define MAX 100

void main(void)
{
#if MAX>99
    printf("compileaza daca matricea este mai mare decit
            99\n");
#endif
}
```


Acest program afişează mesajul pe ecran deoarece **MAX** este mai mare decât 99. Exemplul ilustrează un lucru important. Expresia care urmează după **#if** este evaluată în timpul compilării. De aceea, ea trebuie să conţină doar identificatori şi constante definite anterior - nu pot fi folosite variabile.

Directiva **#else** lucrează asemănător instrucţiunii **else** din limbajul C: ea stabileşte o alternativă dacă **#if** eşuează. Exemplul anterior poate fi extins după cum se arată aici:

```
/* Un exemplu simplu de #if/#else. */
#include <stdio.h>

#define MAX 10

void main(void)
{
#if MAX>99
    printf("compileaza daca matricea este mai mare decat
            99\n");
#else
    printf("compileaza matrice mici\n");
#endif
}
```

În acest caz, **MAX** este definit ca fiind mai mic decât 99, astfel încât porţiunea de cod **#if** nu este compilată. În schimb, va fi compilată alternativa **#else** şi va fi afişat mesajul **compileaza matrice mici**.

Reţineţi că **#else** este folosit pentru a marca atât sfârşitul blocului **#if**, cât şi începutul blocului **#else**. Lucrul aceasta este necesar deoarece poate exista numai un singur **#endif** asociat unui **#if**.

Directiva **#elif** înseamnă „altfel dacă" şi stabileşte lanţul if-else-if (dacă-altfel-dacă) pentru opţiuni de compilare multiple. **#elif** este urmat de o expresie constantă. Dacă expresia este adevărată, este compilat acel bloc de cod şi nici o altă expresie **#elif** nu mai este testată. Altfel, este verificat următorul bloc din serie. Forma generală a directivei **#elif** este:

#if *expresie*
    secvenţă de instrucţiuni
#elif *expresie 1*

secvenţă de instrucţiuni
#elif *expresie 2*
secvenţă de instrucţiuni
#elif *expresie 3*
secvenţă de instrucţiuni
#elif *expresie 4*
secvenţă de instrucţiuni
.
.
.
#elif *expresie N*
secvenţă de instrucţiuni
#endif

De exemplu, următorul fragment foloseşte valoarea din **TARA_CURENTA** pentru a stabili moneda.

```
#define US 0
#define ANGLIA 1
#define FRANTA 2
#define TARA_CURENTA US
#if TARA_CURENTA == US
   char moneda[] = "dolar";
#elif TARA_CURENTA == ANGLIA
   char moneda[] = "lira";
#else
   char moneda[] = "franc";
#endif
```

Standardul ANSI C afirmă că **#if** şi **#elif** pot fi imbricate până la opt niveluri. Standardul propus pentru ANSI C++ sugerează că trebuie să fie permise cel puţin 256 niveluri de imbricare. Când sunt imbricate, fiecare **#endif**, **#else** sau **#elif** se asociază cu cel mai apropiat **#if** sau **#elif**. De exemplu, următorul fragment este perfect valid:

```
#if MAX>100
   #if VERSIUNE_SERIALA
        int port=198;
   #elif
        int port=200;
   #endif
#else
   char buffer_iesire[100];
#endif
```


#ifdef şi #ifndef

O altă metodă de compilare condiţionată foloseşte directivele **#ifdef** şi **#ifndef**, care înseamnă „dacă este definit" şi respectiv „dacă nu este definit". Forma generală pentru **#ifdef** este:

#ifdef *nume_macro*
secvenţă de instrucţiuni
#endif

Dacă *nume_macro* a fost definit anterior într-o instrucţiune **#define**, blocul de cod va fi compilat.
Forma generală pentru **#ifndef** este:

#ifndef *nume_macro*
secvenţă de instrucţiuni
#endif

Dacă *nume_macro* nu este definit curent de o instrucţiune **#define**, blocul de cod este compilat.
Atât **#ifdef** cât şi **#ifndef** pot folosi o instrucţiune **#else**, dar nu şi **#elif**. De exemplu,

```
#include <stdio.h>

#define TED 10

void main(void)
{
#ifdef TED
   printf("Salut Ted\n");
#else
   printf("Salut oricui\n");
#endif
#ifndef RADU
   printf("Radu nu este definit\n");
#endif
}
```

va afişa **Salut Ted** şi **Radu nu este definit**. Însă, dacă **TED** nu ar fi fost definit, s-ar fi afişat **Salut oricui** urmat de **Radu nu este definit**.
Puteţi să imbricaţi **#ifdef** şi **#ifndef** pe cel puţin opt niveluri. Standardul propus ANSI C++ sugerează să fie admise cel puţin 256 de niveluri de imbricare.

#undef

Directiva **#undef** elimină o definiţie anterioară a unui nume de macro care îi urmează. Forma generală pentru **#undef** este:

#undef *nume_macro*

De exemplu,

```
#define LUNG 100
#define GROS 100

char matrice[LUNG][GROS];

#undef LEN
#undef GROS
/* în acest moment atat LUNG cat si GROS nu mai sunt
   definite */
```

Atât **LUNG** cât şi **GROS** sunt definite până se întâlnesc instrucţiunile **#undef**.
În principal, **#undef** este folosită pentru a permite numelor de macro să fie localizate doar în acele secţiuni de cod care au nevoie de ele.

Utilizarea operatorului defined

În afară de **#ifdef** mai există şi o a doua cale pentru a determina dacă un nume de macro este definit. Puteţi folosi directiva **#if** împreună cu operatorul din timpul compilării **defined**. Acesta are forma generală:

defined *nume_macro*

Dacă *nume_macro* este definit curent, atunci expresia este adevărată. Altfel, ea este falsă. De exemplu, pentru a determina dacă este definit numele macro **FISIERULMEU**, puteţi folosi oricare dintre aceste două comenzi pentru preprocesor:

```
#if defined FISIERULMEU
```

sau

```
#ifdef FISIERULMEU
```


De asemenea puteţi să precedaţi **defined** cu ! pentru a obţine reciproca acestei condiţii. De exemplu, următorul fragment este compilat doar dacă **DEPANAT** nu este definit.

```
#if !defined DEPANAT
    printf("Versiune finala!\n");
#endif
```

Un motiv pentru utilizarea lui **defined** este acela că el permite ca prezenţa unui nume de macro să fie determinată printr-o instrucţiune **#elif**.

#line

Directiva **#line** modifică conţinutul din **__LINE__** şi **__FILE__**, care sunt identificatori predefiniţi în compilator. Identificatorul **__LINE__** conţine numărul liniei de cod compilate curent. Identificatorul **__FILE__** este un şir care conţine numele fişierului sursă care este compilat. Forma generală pentru **#line** este:

#line *numar "numefisier"*

unde *numar* este un întreg pozitiv şi devine noua valoare pentru **__LINE__**, iar *numefisier*, care este opţional, poate fi orice identificator valid pentru fişier, care devine noua valoare pentru **__FILE__**. **#line** este folosit în primul rând pentru depanare şi aplicaţii speciale.
De exemplu, următorul cod specifică faptul că numărarea liniilor începe cu 100 şi instrucţiunea **printf()** va afişa valoarea 102, deoarece este a treia linie din program după instrucţiunea **#line 100**.

```
#include <stdio.h>

#line 100                    /* reseteaza numaratorul de
                                linii */
void main(void)              /* linia 100 */
{                            /* linia 101 */
   pritf("%d\n", __LINE__);/* linia 102 */
}
```

#pragma

Directiva **#pragma** este o directivă de definire pentru implementări care permite ca acestea să fie transmise compilatorului. De exemplu, un compilator poate avea o opţiune care permite urmărirea execuţiei programului. O astfel de opţiune va fi

specificată atunci printr-o instrucţiune **#pragma**. Pentru detalii şi opţiuni, trebuie să consultaţi manualul utilizatorului.

Operatorii pentru preprocesor # şi

Există doi operatori pentru preprocesor: # şi ##. Aceşti operatori sunt folosiţi împreună cu instrucţiunea **#define**.

Operatorul #, care este denumit în general operatorul *de înşiruire*, transformă argumentul pe care îl precede într-un şir cu ghilimele. De exemplu, să considerăm acest program:

```
#include <stdio.h>

#define facsir(s) # s

void main(void)
{
    printf(facsir(Imi place C++));
}
```

Preprocesorul de C transformă linia:

```
printf(facsir(Imi place C++));
```

în

```
printf("Îmi place C++");
```

Operatorul ##, numit şi operatorul de inserare, concatenează două elemente. De exemplu,

```
#include <stdio.h>

#define concat(a, b) a ## b

void main(void)
{
    int xy = 10;
    printf("%d", concat(x, y));
}
```

Preprocesorul transformă


```
printf("%d", concat(x, y));
```

în

```
printf("%d", xy);
```

Dacă aceste operaţii vi se par stranii, reţineţi că ele nu sunt necesare şi nici folosite în majoritatea programelor în C/C++. Ele există în primul rând pentru a permite preprocesorului să trateze anumite cazuri speciale.

Nume de macro predefinite

C are încorporate cinci nume de macro predefinite. Ele sunt:

__LINE__
__FILE__
__DATE__
__TIME__
__STDC__

Numele macro **__LINE__** şi **__FILE__** au fost discutate în secţiunea despre **#line**.

Numele macro **__DATE__** conţine un şir de forma *lună/zi/an*. El reprezintă data calendaristică a translatării fişierului sursă în cod obiect.

Ora translatării codului sursă în cod obiect este conţinută sub formă de şir în **__TIME__**. Formatul şirului este *ore:minute:secunde*.

__STDC__ conţine constanta în baza 10 1. Aceasta înseamnă că implementarea se conformează standardului ANSI C. Dacă numele macro conţine orice alt număr (sau dacă nu este definit), implementarea diferă de standard sau este un program în C++.



***NOTĂ:** Propunerea de ANSI C++ include anterioarele nume macro predefinite şi adaugă încă unul: **__cplusplus**. Acest macro este definit (ca şi celelalte) când se compilează un program în C++. De aceea, la compilarea unui program în C++, **__cplusplus** va fi definit iar **__STDC__**, în general, nu va fi definit. (Practic, în C++, semnificaţia parametrului **__STDC__** este dependentă de implementare, ceea ce înseamnă că el poate, teoretic, să fie definit la compilarea unui program în C++.)*

Comentarii

În C toate comentariile încep cu perechea de caractere /* şi se încheie cu */. Nu trebuie să existe spaţii între asterisc şi slash. Compilatorul ignoră orice text între

începutul şi sfârşitul simbolurilor de comentariu. De exemplu, următorul program afişează pe ecran doar va.

```
#include <stdio.h>

void main(void)
{
    printf("va");
    /* printf("salut"); */
}
```

Comentariile pot fi plasate oriunde în program, atât timp cât ele nu apar în mijlocul unui cuvânt-cheie sau identificator. Aceasta înseamnă că următorul comentariu este valid:

```
x = 10+ /*aduna numere */5;
```

în timp ce

```
swi/*asta nu va lucra*/tch(c) { ...
```

este incorect, deoarece un cuvânt-cheie nu poate conţine un comentariu. Totuşi, în general nu ar trebui să plasaţi comentarii în mijlocul expresiilor deoarece le vor face conţinutul de neînţeles.

Comentariile nu pot fi imbricate. Aceasta înseamnă că un comentariu nu poate să conţină un altul. De exemplu, următorul fragment de cod determină o eroare în timpul compilării.

```
/* acesta este un comentariu exterior
    x = y/a;
    /* acesta este un comentariu interior - si determina o
        eroare */
*/
```

Ar trebui să includeţi comentarii ori de câte ori sunt necesare pentru a explicita operaţia din cod. Toate funcţiile, în afara celor evidente, ar trebui sa fie precedate de un comentariu care să arate ce fac, cum sunt apelate şi ce returnează.

NOTĂ: C++ admite în întregime stilul de comentarii din C. În plus, el vă permite să definiţi un comentariu de un rând. Acesta începe cu // şi se încheie la sfârşitul rândului.

Partea a II-a

C++ - Caracteristici specifice

Partea a doua a acestei cărţi examinează caracteristicile specifice ale limbajului C++. Aceasta înseamnă că în ea se discută acele caracteristici ale lui C++ care nu au nimic comun cu C. (Caracteristicile de tip C din C++ sunt descrise în Partea întâi.) C++ este, în esenţă, un super C, astfel încât aproape tot ceea ce cunoaşteţi despre C este aplicabil în C++. Deoarece majoritatea îmbunătăţirilor din C++ faţă de C sunt proiectate pentru a admite programarea orientată pe obiecte (OOP), Partea a doua mai asigură şi o discuţie despre teoria şi calităţile acestui tip de programare.

NOTĂ: Această parte presupune că ştiţi să programaţi în C. Cunoaşterea limbajului C este o cerinţă obligatorie pentru învăţarea lui C++, astfel încât, dacă nu cunoaşteţi încă C, rezervaţi-vă mai întâi timp pentru a-l învăţa pe acesta.

Capitolul 11

O privire de ansamblu asupra lui C++

Acest capitol prezintă o privire de ansamblu asupra conceptelor esenţiale pe care se bazează C++. C++ este un limbaj de programare orientat pe obiecte, iar caracteristicile sale de orientare pe obiecte sunt în mare măsură corelate între ele. În numeroase situaţii această interdependenţă face dificil de descris o caracteristică de C++ fără implicarea multor altora. În anumite cazuri, caracteristicile orientate pe obiecte ale limbajului C++ sunt atât de împletite încât discutarea uneia presupune cunoştinţe anterioare despre una sau mai multe alte caracteristici. De aceea, acest capitol prezintă o scurtă privire de ansamblu a celor mai importante aspecte din C++. Următoarele capitole ale acestei părţi examinează în detaliu C++.

Originile limbajului C++

După cum ştiţi, C++ este o versiune îmbogăţită a lui C. Extensiile lui C++ faţă de C au fost create de către Bjarne Stroustrup în 1980 în Laboratoarele Bell din Murray Hill, New Jersey. Iniţial, el a numit noul limbaj „C cu clase". Dar, în 1983, numele a fost schimbat în C++.

Chiar dacă predecesorul său, C, este unul dintre cele mai îndrăgite şi mai larg utilizate limbaje de programare profesionale din lume, inventarea lui C++ a fost impusă de o importantă cerinţă a programării: creşterea complexităţii. În timp, programele pentru calculator au devenit mai mari şi mult mai complexe. Chiar dacă C este un excelent limbaj de programare, încă mai are limitele sale. În C, dacă un program trece de la 25.000 la 100.000 de linii de cod devine atât de complex încât este dificil să fie stăpânit ca un întreg. Scopul lui C++ este să depăşească această barieră. Esenţa sa este să permită programatorului să înţeleagă şi să administreze programe mai mari şi mult mai complexe.

Majoritatea îmbunătăţirilor aduse de Stroustrup pentru C admite programarea orientată pe obiecte, uneori numită şi OOP. (A se vedea paragraful următor pentru o scurtă explicare a programării orientate pe obiecte.) Stroustrup afirmă că unele dintre caracteristicile de orientare pe obiecte au fost inspirate dintr-un alt astfel de limbaj, numit Simula67. Astfel, C++ reprezintă combinarea a două metode de programare puternice.

De la apariţie, C++ a trecut prin trei mari revizuiri, una în 1985, alta în 1989, iar a treia când a început lucrul la standardul ANSI pentru C++. Prima versiune a standardului propus a fost creată pe 25 ianuarie 1994. Comitetul ANSI C++ (al cărui membru sunt) a păstrat virtual toate caracteristicile definite iniţial de Stroustrup şi, de asemenea, a adăugat multe altele noi.

Procesul de standardizare este tipic unul lent şi vor trece ani până când va fi adoptat standardul pentru C++. De aceea, ţineţi minte că C++ este încă „în lucru" şi că unele caracteristici urmează să fie puse la punct. Totuşi, materialul prezentat în această carte este stabil. El este aplicabil, de asemenea, tuturor compilatoarelor de C++ existente şi este în concordanţă cu standardul propus pentru ANSI C++.

Când a inventat C++, Stroustrup ştia că este important să păstreze spiritul iniţial al limbajului C, inclusiv eficienţa, flexibilitatea şi concepţia sa de bază, că programatorul conduce jocul, şi nu limbajul, adăugându-i în acelaşi timp şi suportul pentru programare orientată pe obiecte. Din fericire, scopurile sale au fost atinse: C++ asigură programatorului libertatea şi controlul din C, împreună cu puterea obiectelor. Pentru a folosi cuvintele lui Stroustrup, caracteristicile de orientare pe obiecte din C++ „permit programelor să fie structurate pentru a fi clare, extensibile şi uşor de întreţinut, fără pierderea eficienţei."

Chiar dacă C++ a fost iniţial proiectat pentru a fi de folos în administrarea programelor foarte mari, nu există o limitare a utilizării sale. De fapt, atributele sale de orientare pe obiecte pot fi aplicate potenţial tuturor sarcinilor de programare. Nu este neobişnuit să vedem C++ folosit pentru obiecte cum ar fi editoare, baze de date, fişiere de personal şi programe de comunicare. De asemenea, deoarece C++ preia eficienţa lui C, cu ajutorul lui s-au creat multe sisteme de înaltă performanţă.

Ce este programarea orientată pe obiecte?

Programarea orientată pe obiecte (OOP) este o nouă cale de abordare a programării. Modalităţile de programare s-au schimbat imens de la inventarea calculatorului, în primul rând pentru a se acomoda creşterii complexităţii programelor. De exemplu, la început, când au fost inventate calculatoarele, programarea se făcea introducându-se instrucţiuni în maşina în cod binar cu ajutorul panoului frontal al calculatorului. Acest lucru a fost convenabil atât timp cât programele aveau doar câteva sute de instrucţiuni. O dată cu mărirea programelor, au fost inventate limbajele de asamblare, astfel încât programatorii se puteau descurca cu programe mai mari, cu complexitate crescută, folosind reprezentarea simbolică a instrucţiunilor pentru maşină. Deoarece programele continuau să crească, au fost introduse limbajele de nivel înalt care oferă programatorului mai multe unelte cu care să facă faţă complexităţii. Primul limbaj larg răspândit a fost, desigur, FORTRAN. Chiar dacă el a fost un prim pas foarte impresionant, este departe de a fi un limbaj care încurajează programe clare, uşor de înţeles.

Anii '60 au dat naştere programării structurate. Aceasta este metoda încurajată de limbaje cum sunt C şi Pascal. Utilizarea limbajelor structurate face posibilă scrierea destul de uşoară a unor programe relativ complexe. Totuşi, chiar folosind metodele programării structurate, un proiect nu mai poate fi controlat o dată ce atinge anumite mărimi (adică o dată ce complexitatea sa o depăşeşte pe cea pe care o poate controla un programator).

Luaţi în calcul că pentru fiecare realizare din dezvoltarea programării au fost create metode care să permită programatorului să se descurce cu complexitate crescută. La fiecare pas al drumului noua abordare a preluat cele mai bune elemente ale metodelor anterioare şi a continuat drumul. Astăzi multe proiecte sunt

aproape sau în punctul în care programarea structurată nu mai face față. Pentru a rezolva această problemă, a fost inventată programarea orientată pe obiecte.

Programarea orientată pe obiecte a preluat cele mai bune idei ale programării structurate şi le combină cu mai multe concepte noi, mai puternice, care vă încurajează să abordați sarcina programării într-un mod nou. În general, când programați în modul orientat pe obiecte, împărțiți o problemă în subgrupe de secțiuni înrudite, care țin seama atât de codul cât şi de datele corespunzătoare din fiecare grup. Apoi, organizați aceste subgrupe într-o structură ierarhică. În sfârşit, le transformați în unități de sine stătătoare numite obiecte.

Toate limbajele de programare orientate pe obiecte au trei caracteristici comune: încapsulare, polimorfism şi moştenire. Să le examinăm pe fiecare, pe scurt.

Încapsularea

Încapsularea este un mecanism care leagă împreună cod şi date şi le păstrează pe ambele în siguranță față de intervenții din afară şi de utilizări greşite. Mai mult, încapsularea este cea care permite crearea unui obiect. Spus simplu, un *obiect* este o entitate logică ce încapsulează atât date cât şi cod care manevrează aceste date. Într-un obiect o parte din cod şi/sau date pot fi particulare acelui obiect şi inaccesibile pentru orice din afara sa. În acest fel, un obiect dispune de un nivel semnificativ de protecție care împiedică modificarea accidentală sau utilizarea incorectă a părților proprii obiectului de către secțiuni ale programului cu care nu are legătură.

În cele din urmă, un obiect este o variabilă de un tip definit de utilizator. La început poate să pară ciudat să considerăm un obiect, care leagă atât cod cât şi date, ca fiind o variabilă. Totuşi, în programarea orientată pe obiecte aşa stau lucrurile. Când definiți un obiect, implicit creeați un nou tip de date.

Polimorfism

Limbajele de programare orientate pe obiecte admit *polimorfismul*, care este caracterizat prin fraza „o interfață, metode multiple". Mai clar, polimorfismul este caracteristica ce permite unei interfețe să fie folosită cu o clasă generală de acțiuni. Acțiunea specifică selectată este determinată de natura precisă a situației. Un exemplu din practica zilnică pentru polimorfism este un termostat. Nu are importanță ce combustibil întrebuințați pentru încălzirea casei (gaze, petrol, electricitate etc.), termostatul lucrează în acelaşi fel. În acest caz, termostatul (care este interfața) este acelaşi indiferent de combustibil (metodă). De exemplu, dacă doriți temperatura de 22 de grade, veți regla termostatul la 22 de grade. Nu are importanță combustibilul care produce căldura. Acelaşi principiu se poate aplica şi programării. De exemplu, puteți avea un program care defineşte trei tipuri de memorie stivă. Una este folosită pentru valori întregi, una pentru valori tip caracter şi una pentru valori în virgulă mobilă. Datorită polimorfismului, puteți crea trei perechi de funcții numite **pune()** şi **scoate()** - câte una pentru fiecare tip de date. Conceptul general (interfața) este cel de a pune şi de a scoate date dintr-o memorie stivă. Funcțiile definesc calea specifică (metoda) care se foloseşte pentru fiecare tip de date. Când puneți date în memoria stivă, tipul de date va fi cel care va determina versiunea particulară a lui **pune()** care va fi apelată. (Veți vedea în curând un asemenea exemplu.)

Polimorfismul ajută la reducerea complexității permițând aceleiaşi interfețe să fie folosită pentru a specifica o clasă generală de acțiuni. Rolul compilatorului este să aleagă *acțiunea specifică* (deci, metoda) care se aplică fiecărei situații. Dvs., programatorul, nu trebuie să executați personal această acțiune. Nu trebuie decât să vă amintiți şi să folosiți interfața generală.

Primele limbaje de programare orientate pe obiecte au fost interpretoarele, astfel încât polimorfismul a fost admis, desigur, în timpul rulării. Dar C++ este un limbaj de compilare. Astfel, în C++, polimorfismul este admis atât în timpul rulării cât şi în timpul compilării.

Moştenirea

Moştenirea este procesul prin care un obiect poate să preia prototipul altui obiect. Acest lucru este important deoarece se admite conceptul de *clasificare*. Dacă vă gândiți puțin, majoritatea cunoştințelor sunt accesibile deoarece sunt clasificate ierarhic. De exemplu, un măr ionatan face parte din clasificarea *măr*, care la rândul său face parte din clasa *fructe*, care se află în marea clasă a *hranei*. Fără utilizarea claselor, fiecare obiect ar trebui definit explicitându-se toate caracteristicile sale. Însă, prin folosirea clasificărilor, un obiect are nevoie doar de definirea acelor calități care îl fac unic în clasa sa. Mecanismul moştenirii este acela care face posibil ca un obiect să fie un exemplar specific al unui caz mai general. După cum veți vedea, moştenirea este un aspect important al programării orientate pe obiecte.

Programarea în stilul C++

Deoarece C++ este un super C, puteți să scrieți programe în C++ care să arate exact ca şi cele din C. Totuşi, făcând aşa, vă lipsiți de a avea toate avantajele limbajului C++. (Este ca şi cum ați privi la un televizor color cu butonul de culoare la minim!) Astfel, cei mai mulți programatori în C++ folosesc stilul şi anumite caracteristici care sunt unice în C++. Majoritatea diferențelor de stil între programele în C şi cele în C++ se regăseşte în avantajul pe care îl are C++ prin capacitățile de orientare pe obiecte. Dar un alt avantaj al folosirii stilului de programare propriu lui C++ este acela că vă ajută să gândiți în C++, şi nu în C. (Asta înseamnă că, adoptând un stil diferit când scrieți coduri în C++, vă veți spune să nu mai gândiți în C şi să începeți să gândiți în C++.)

Deoarece este important să învățați să scrieți programe în C++ care să *arate* ca fiind în C++, acest paragraf introduce câteva din caracteristicile necesare. Examinați acest program în C++:

```
#include <iostream.h>

main()
{
    int i;

    cout << "Iata iesirea.\n"; // un comentariu de o linie
    /* puteti inca sa folositi comentariile in stilul C */

    // introduceti un numar folosind >>
    cout << " introduceti un numar:";
    cin >> i;

    // acum scrieti un numar folosind <<
    cout << i << " la patrat este " << i*i << "\n";
    return 0;
}
```

După cum puteți vedea, acest program arată mult diferit față de programele din Partea întâi. La început este inclus fișierul antet IOSTREAM.H. El este definit de C++ și folosit pentru a admite operațiile de I/O în stilul C++. (IOSTREAM.H este pentru C++ ceea ce este STDIO.H pentru C.)

Prima modificare stilistică se află în această linie:

```
main()
```

Rețineți că lista de parametri din **main()** este vidă. În C++, aceasta indică faptul că **main()** nu are parametri, spre deosebire de C, unde o funcție care nu are parametri trebuie să folosească **void** în lista sa de parametri, așa cum se arată aici:

```
main(void)
```

Totuși, în C++ utilizarea argumentului **void** este superfluă și inutilă. Ca regulă generală, când o funcție nu are parametri în C++, lista sa de parametri este pur și simplu goală; folosirea argumentului **void** nu este necesară.

Următoarea diferență se află pe linia aceasta:

```
cout << "Iata iesirea.\n";  // un comentariu de o linie
```


Această linie introduce două noi caracteristici din C++. Prima, instrucțiunea

```
cout << "Iata iesirea.\n";
```

face să fie afișată pe ecran **Iata iesirea.**, urmată de o combinație început de rând/linie nouă. În C++, **<<** are un rol extins. El este încă operatorul de deplasare la stânga, dar, când este folosit ca în exemplul prezentat, este, totodată, un *operator de ieșire*. Cuvântul **cout** este un specificator în legătură cu ecranul. (De fapt, ca și C, C++ admite indirectarea I/O, dar în cadrul acestei discuții, să presupunem că **cout** se referă la ecran.) Puteți să utilizați **cout** și **<<** pentru a produce la ieșire orice tipuri de date încorporate în C++, precum și șiruri de caractere.

Rețineți că puteți să folosiți în continuare **printf()** sau oricare alte funcții de I/O din C în programele în C++. (Desigur că trebuie să includeți STDIO.H dacă doriți să folosiți funcțiile de I/O din C.) Dar majoritatea programatorilor simt că utilizând **cout <<** este mult mai aproape de spiritul C++. Mai mult, chiar dacă folosirea funcției **printf()** pentru ieșirea șirului este virtual echivalentă în acest caz cu utilizarea operatorului **<<**, sistemul de I/O din C++ poate să se extindă pentru a efectua operații asupra obiectelor pe care le definiți (ceea ce nu poate face **printf()**).

Ceea ce urmează expresiei pentru ieșire este un comentariu tip C++. În C++ comentariile sunt definite în două moduri. În primul rând puteți folosi un comentariu tip C, care lucrează în C++ la fel ca în C. Însă, în C++ puteți să mai definiți un *comentariu cu o singură linie* folosind //. Când începeți un comentariu cu //, orice urmează este ignorat de compilator până la sfârșitul liniei. În general, programatorii în C++ folosesc comentariile tip C când creează comentarii cu mai multe linii iar pe cele tip C++ când este suficientă una singură.

În continuare, programul solicită utilizatorului un număr. Numărul este citit de la tastatură cu următoarea instrucțiune:

```
cin >> i;
```

În C++ operatorul **>>** își mai menține semnificația de deplasare la dreapta. Dar, când este utilizat așa cum s-a arătat, el este și *operatorul de intrare* pentru C++. Această instrucțiune determină ca lui **i** să i se dea o valoare citită de la tastatură. Specificatorul **cin** se referă la echipamentul de intrare standard, care, de obicei, este tastatura. În general, puteți să folosiți **cin >>** pentru a introduce o variabilă de oricare tip de bază, inclusiv una de tip șir.

__NOTĂ:__ Linia de cod descrisă anterior nu este tipărită greșit. Mai ales nu trebuie să presupuneți că ar fi trebuit să existe & în fața lui __i__. După cum știți, când introduceți informații folosind o funcție precum __scanf()__, trebuie să transmiteți explicit un pointer către variabila care va primi informația. Aceasta înseamnă să

*precedați numele variabilei cu operatorul „adresa lui", &. Totuși, prin modul în care este definit operatorul >> în C++, nu este necesar (de fapt nu trebuie) să folosiți operatorul &. În **Capitolul 17** veți învăța de ce este așa.*

Chiar dacă nu este ilustrat printr-un exemplu, aveți libertatea să utilizați orice funcție de intrare din C, cum ar fi **scanf()**, în loc să folosiți **cin>>**. Totuși, ca și pentru **cout**, majoritatea programatorilor simt că **cin** este mai aproape de spiritul limbajului C++.

O altă linie interesantă din program este următoarea:

```
cout << i << "la patrat este " << i*i << "\n";
```

Presupunând că **i** are valoarea 10, această instrucțiune determină să se afișeze **10 la patrat este 100**, urmat de început de rând / linie nouă. Așa cum ilustrează această linie, puteți asocia mai multe operații de ieșire <<.

Observați că programul se încheie cu această instrucțiune:

```
return 0;
```

Ea determină returnarea unui zero către procesul care a făcut apelarea (care este, de obicei, sistemul de operare). Returnarea lui zero indică terminarea normală a programului. Terminarea sa anormală trebuie semnalată prin returnarea unei valori diferite de zero.

Există două căi prin care **main()** este declarată uzual într-un program în C++: fie ca nedeclarând nimic (ceea ce înseamnă că tipul returnat este **void**), fie ca returnând o valoare întreagă. Țineți minte că nu este necesar ca **main()** să returneze o valoare. De fapt, majoritatea programelor din **Partea întâi** a acestei cărți declară simplu **main()** ca fiind **void**. În **Partea a doua**, **main()** va returna o valoare doar pentru a ilustra cele două forme uzuale. Oricum, în programele dvs. puteți utiliza oricare dintre metode.

O privire mai atentă asupra operatorilor de I/O

Cum s-a mai spus, când sunt folosiți pentru I/O, operatorii << și >> sunt capabili să manevreze orice tip de date încorporat în C++. De exemplu, acest program introduce o variabilă **float**, una **double** și un șir, apoi le afișează:

```
#include <iostream.h>

main()
{
    float f;
    char sir[80];
```



```
    double d;

    cout << "Introduceti doua numere in virgula mobila: ";
    cin >> f >> d;

    cout << "Introduceti un sir: ";
    cin >> sir;

    cout << f << " " << d << " " << sir;
    return 0;
}
```

Când rulați acest program, încercați, la solicitarea unui șir, să introduceți **Acesta este un test**. Când programul va reafișa informația pe care ați introdus-o, va fi afișat doar cuvântul „Acesta". Restul șirului nu este afișat deoarece operatorul >> lucrează cu șirurile în același fel în care o face și specificatorul **%s** pentru **scanf()**. El încheie citirea intrării când este întâlnit primul caracter de spațiu liber. De aceea, „este un test" nu va fi citit niciodată de către program.

Acest program ilustrează, de asemenea, faptul că puteți să înșiruiți câteva operații de intrare într-o singură instrucțiune.

Declararea variabilelor locale

O altă diferență între cum scrieți codul de C și cel de C++ este locul în care pot fi declarate variabilele locale. În C, trebuie să declarați toate variabilele locale dintr-un bloc la începutul acestuia. Nu puteți să declarați o variabilă locală într-un bloc după ce a apărut o instrucțiune de „acțiune". De exemplu, în C, acest fragment este incorect:

```
/* Incorect in C. OK în C++. */
f()
{
    int i,k;

    for(i=0; i<10; i++) {
          k = i+1;
          int j; /* nu va fi compilat ca program in C*/
          j = i*2;
          .
          .
          .
    }
}
```

Deoarece o instrucţiune de „acţiune" precede declararea lui **j**, un compilator C va afişa o eroare şi va refuza să compileze această funcţie. Dar, în C++, acest fragment este perfect acceptabil şi va fi compilat fără eroare. Puteţi să declaraţi variabile în C++ în orice punct dintr-un bloc - nu doar la început.

Iată o altă versiune a programului din secţiunea precedentă, în care fiecare variabilă este declarată atunci când este necesară:

```
#include <iostream.h>

main()
{
    float f;
    double d;

    cout << "Introduceti doua numere in virgula mobila: ";
    cin >> f >> d;

    cout << "Introduceti un sir: ";
    char sir[80];
    // sir este declarat aici, chiar inainte de prima folosire
    cin >> sir;

    cout << f << " " << d << " " << sir;
    return 0;
}
```

Este la latitudinea dvs. să declaraţi toate variabilele la începutul blocului sau în momentul primei utilizări. Deoarece concepţia lui C++ se bazează pe încapsularea codului şi a datelor, este normal să declaraţi variabilele aproape de momentul folosirii şi nu la începutul blocului. În exemplul precedent declaraţiile sunt separate doar demonstrativ. Dar, este uşor să imaginaţi exemple mult mai complexe în care această caracteristică a limbajului C++ să fie mai evidentă.

Declararea variabilelor în apropierea locului unde le veţi utiliza vă ajută să evitaţi efectele secundare accidentale. Însă, marele avantaj al declarării variabilelor în momentul primei utilizări apare la funcţiile mari. Cinstit, în funcţiile scurte (ca multe exemple din această carte), sunt puţine motive pentru a nu declara toate variabilele la începutul lor. Din acest motiv, în carte variabilele se vor declara în momentul primei utilizări doar când vor exista justificări legate de mărimea sau de complexitatea unei funcţii.

Există unele neînţelegeri în ce priveşte motivaţia generală a localizării variabilelor. Oponenţii susţin că împrăştierea declaraţiilor prin tot blocul face mai grea, nu mai uşoară, citirea de către cineva a codului pentru a afla rapid declararea tuturor variabilelor folosite în el, iar programul devine mai greu de

întreţinut. Din acest motiv, unii programatori în C++ nu utilizează des această facilitate. Cartea nu adoptă nici o poziţie în această problemă. Totuşi, când este aplicată corect, în special în funcţii mari, declararea variabilelor în punctul primei lor utilizări poate să vă ajute să creaţi mai uşor programe fără eroare.

Prezentarea claselor C++

Acest paragraf prezintă cea mai importantă caracteristică din C++: clasele. Pentru a crea un obiect în C++ trebuie să îi definiţi mai întâi forma sa generală folosind cuvântul cheie **class**. O **class** este similară sintactic cu o structură. Următoarea clasă defineşte un tip numit **stiva**, care este utilizat pentru a crea o memorie stivă:

```
#define SIZE 100

// Aceasta creeaza clasa numita stiva.
class stiva {
    int stiv[SIZE];
    int tos;
public:
    void init();
    void pune(int i);
    int scoate();
};
```

O clasă poate conţine atât secţiuni particulare cât şi publice. Implicit, toate elementele definite într-o clasă sunt particulare. De exemplu, variabilele **stiv** şi **tos** sunt particulare. Aceasta înseamnă că o funcţie care nu face parte din acea clasă nu poate avea acces la ele. Iată unul dintre modurile de realizare a încapsulării - accesul la anumite elemente poate fi strict controlat păstrându-le particulare. Chiar dacă nu apare în exemplu, puteţi să definiţi, de asemenea, funcţii particulare, care apoi pot fi apelate doar de alţi membri ai clasei.

Pentru a crea părţi publice ale unei clase (accesibile altor părţi ale programului), trebuie să le declaraţi după cuvântul cheie **public**. La toate variabilele sau funcţiile definite după **public** pot să aibă acces toate celelalte funcţii din program. În esenţă, restul programului are acces la un obiect prin funcţiile sale publice. Ar trebui menţionat acum că, deşi puteţi avea variabile publice, teoretic ar trebui să limitaţi sau să eliminaţi utilizarea lor. Ar trebui să faceţi toate datele particulare iar accesul pentru a le controla să-l permiteţi prin funcţii publice. Reţineţi: cuvântul cheie **public** este urmat de două puncte.

Funcţiile **init()**, **pune()** şi **scoate()** sunt denumite *funcţii membre* deoarece ele fac parte din clasa **stiva**. Variabilele **stiv** şi **tos** sunt denumite *variabile membre* (sau *date membre*). Amintiţi-vă că un cod este o grupare de cod şi de date. Doar

funcţiile membre au acces la membrii particulari ai clasei în care sunt declarate. De aceea, doar **init()**, **pune()** şi **scoate()** pot să aibă acces la **stiv** şi **tos**.

O dată ce aţi definit o clasă, puteţi să creaţi un obiect de acel tip folosind numele clasei. În esenţă, numele clasei devine un nou specificator de tip de date. De exemplu, linia următoare creează un obiect numit **stivamea** de tip **stiva**.

```
stiva stivamea;
```

Puteţi, de asemenea, să creaţi variabile când este definită o clasă punând numele lor după închiderea acoladei, exact în acelaşi fel în care aţi face-o cu o structură.

Să recapitulăm: în C++ **class** creează un nou tip de date care pot fi folosite pentru a construi obiecte de acel tip. De aceea, un obiect este un exemplar (o instanţiere) al unei clase exact în acelaşi fel în care altă variabilă este, de exemplu, un exemplar al tipului de date **int**. Altfel spus, o clasă este o abstractizare logică, iar un obiect este real. (Asta înseamnă că un obiect există în memoria calculatorului.)

Forma generală a declarării unei clase este:

class *nume-clasa* {
    date şi funcţii particulare
public:
    date şi funcţii publice
}   *lista de obiecte*;

Desigur, *lista de obiecte* poate fi goală.

În interiorul declaraţiei pentru **stiva**, elemente de tip funcţie au fost specificate folosindu-se prototipurile lor. În C++, toate funcţiile trebuie să aibă prototipuri. Ele nu sunt opţionale.

Când vine momentul să scrieţi efectiv codul unei funcţii care este membru al unei clase, trebuie să spuneţi compilatorului cărei clase aparţine aceasta, specificând înaintea numelui său numele clasei al cărei membru este. De exemplu, iată un mod de a scrie funcţia **pune()**:

```
void stiva::pune(int i)
{
    if(vis==SIZE) {
        cout << "Stiva este plina. ";
        return;
    }
    stiv[vis] = i;
    vis++;
}
```


:: este numit *operatorul de specificare a domeniului*. Important este că el îi spune compilatorului că această versiune a funcţiei **pune()** aparţine clasei **stiva**, sau, altfel spus, funcţia **pune()** face parte din domeniul **stiva** . După cum veţi vedea, mai multe clase diferite pot să folosească acelaşi nume de funcţie. Compilatorul ştie care funcţie aparţine fiecărei clase datorită operatorului de specificare a domeniului.

Când vă referiţi la un membru al unei clase dintr-o secţiune de cod care nu face parte din acea clasă, trebuie întotdeauna să o faceţi în legătură cu un obiect al acelei clase. Pentru aceasta, folosiţi numele obiectului urmat de operatorul punct şi de numele acelui membru. Regula se aplică ori de câte ori aveţi acces la o funcţie membru sau la date membre. De exemplu, următoarele instrucţiuni apelează **init()** pentru obiectul **stiva**:

```
stiva stival, stiva2;
stival.init();
```

Acest fragment creează două obiecte (**stiva1** şi **stiva2**) şi iniţializează **stiva1**. În acest punct este foarte important să înţelegeţi că **stiva1** şi **stiva2** sunt două obiecte distincte. Aceasta înseamnă că, de exemplu, iniţializarea lui **stiva1** *nu* determină şi iniţializarea lui **stiva2**. Singura legătură dintre **stiva1** şi **stiva2** este aceea că ele sunt obiecte de acelaşi tip.

O funcţie membru poate să apeleze un alt asemenea membru sau să se refere la nişte date membre direct, fără utilizarea operatorului punct. Numele obiectului şi operatorul punct trebuie folosite doar atunci când un cod care nu aparţine clasei apelează un membru.

Programul prezentat aici alătură toate secvenţele şi detaliile care lipseau şi ilustrează clasa **stiva**;

```
#include <iostream.h>

#define SIZE 100

// Aceasta creeaza clasa stiva.
class stiva {
    int stiv[SIZE];
    int vis;
public:
    void init();
    void pune(int i);
    int scoate();
};

void stiva::init()
```

```
{
    vis = 0;
}

void stiva::pune(int i)
{
    if(vis==SIZE) {
          cout << "Stiva este plina.";
          return;
    }

    stiv[vis] = i;
    vis++;
}

int stiva::scoate()
{
    if(vis==0) {
          cout << "Depasire inferioara.";
          return 0;
    }
    vis--;
    return stiv[vis];
}

main()
{
    stiva stiva1, stiva2; // creeaza doua obiecte "stiva"

    stiva1.init();
    stiva2.init();

    stiva1.pune(1);
    stiva2.pune(2);

    stiva1.pune(3);
    stiva2.pune(4);

    cout << stiva1.scoate() << " ";
    cout << stiva1.scoate() << " ";
    cout << stiva2.scoate() << " ";
    cout << stiva2.scoate() << "\n";
```



```
    return 0;
}
```



***REŢINEŢI:** Elementele particulare ale unui obiect sunt accesibile doar funcţiilor care sunt membre ale acelui obiect. De exemplu, o instrucţiune ca aceasta:*

```
stiva1.vis = 0; // eroare
```

*nu poate exista în interiorul funcţiei **main()** a programului precedent deoarece **vis** este particular.*

Prin convenţie, majoritatea programatorilor în C++ pun ca primă funcţie din program funcţia **main()**. Totuşi, în programul pentru **stiva**, funcţiile membre sunt definite înaintea funcţiei **main()**. Deşi nu există o regulă care să impună acest lucru (ele pot fi definite oriunde în program), este cel mai uzual mod de a scrie codul de C++. (Totuşi, funcţiile care nu sunt membre sunt definite ca de obicei, după **main()**. Această carte va urma această convenţie. Desigur, în aplicaţiile reale, clasele asociate unui program vor fi conţinute în fişierul antet.

Funcţii supraîncărcate (overload)

Un mod în care C++ realizează polimorfismul este acela de supraîncărcare a funcţiilor (overloading). În C++, două sau mai multe funcţii pot să aibă acelaşi nume atât timp cât declaraţiile lor de parametri sunt diferite. În această situaţie, se spune că funcţiile cu acelaşi nume sunt supraîncărcate iar procesul este numit *supraîncărcarea funcţiilor.*

Pentru a vedea de ce sunt importante funcţiile supraîncărcate, să considerăm pentru început trei funcţii care se găsesc, probabil, în biblioteca standard a tuturor compilatoarelor de C/C++: **abs()**, **labs()** şi **fabs()**. Funcţia **abs()** returnează valoarea absolută a unui întreg, **labs()** returnează valoarea absolută a unei variabile **float**, iar **fabs()** pe cea a unei variabile **double**. Chiar dacă aceste funcţii efectuează acţiuni aproape identice, în C trebuie să fie folosite toate trei numele, puţin diferite, pentru a reprezenta aceste sarcini, similare în esenţă. Conceptual, situaţia devine mai complexă decât este de fapt. Deşi ideea de bază a fiecărei funcţii este aceeaşi, programatorul trebuie să-şi amintească trei lucruri, nu doar unul singur. În C++, însă, puteţi folosi doar un singur nume pentru toate trei funcţiile, aşa cum ilustrează următorul program:

```
#include <iostream.h>

// abs este definita in trei feluri
int abs(int i);
```

```
double abs(double d);
long abs(long l);

main()
{
    cout << abs(-10) << "\n";

    cout << abs(-11.0) << "\n";

    cout << abs(-9L) << "\n";

    return 0;
}

int abs(int i)
{
    cout << "utilizarea abs() pentru intregi\n";
    return i<0 ? -i : i;
}

double abs(double d)
{
    cout << "utilizarea abs() pentru double\n";
    return d<0 ? -d : d;
}

long abs(long l)
{
    cout << "utilizarea abs() pentru long\n";
    return l<0 ? -l : l;
}
```

Acest program creează trei funcţii asemănătoare, dar totuşi diferite, numite **abs()**, fiecare din ele returnând valoarea absolută pentru argumentul său. Compilatorul ştie ce funcţie să apeleze în fiecare situaţie datorită tipului argumentului. Importanţa funcţiilor supraîncărcate este aceea că ele permit ca seturi de funcţii înrudite să fie accesibile printr-un singur nume. De aceea, numele **abs()** reprezintă *acţiunea generală* care este efectuată. Este lăsat pe seama compilatorului să aleagă *metoda particulară* corectă pentru situaţii specifice. Programatorul trebuie să-şi amintească doar acţiunea generală care trebuie efectuată. Datorită polimorfismului, se reduce numărul de lucruri care trebuie reţinute de la trei la unul. Acest exemplu este aproape banal, dar dacă extindeţi conceptul, puteţi vedea cum polimorfismul poate să vă ajute să trataţi programele foarte complexe.

În general, pentru a supraîncărca o funcţie, declaraţi pur şi simplu diferite versiuni ale ei. De restul are grijă compilatorul. Când supraîncărcaţi o funcţie, trebuie să reţineţi o restricţie importantă: tipul şi/sau numărul de parametri al fiecărei funcţii supraîncărcate trebuie să difere. Nu este suficient pentru două funcţii să difere doar prin tipul returnat de ele. Ele trebuie să difere prin tipul sau numărul de parametri. (Tipul returnat nu asigură suficiente informaţii în toate cazurile pentru ca un compilator să decidă ce funcţie să folosească.) Desigur, funcţiile supraîncărcate *pot* să difere şi prin tipul returnat.

Iată alt exemplu care utilizează funcţii supraîncărcate:

```
#include <iostream.h>
#include <stdio.h>
#include <string.h>

void adunsir(char *s1, char *s2);
void adunsir(char *s1, int i);

main()
{
    char sir[80];

    strcpy(sir, "Va ");
    adunsir(sir, "salut");
    cout << sir <<"\n";

    adunsir(sir, 100);
    cout << sir << "\n";

    return 0;

}
// concateneaza doua siruri
void adunsir(char *s1, char *s2)
{
    strcat(s1, s2);
}

// concateneaza un sir cu intreg "sir"
void adunsir(char *s1, int i)
{
    char temp[80];

    sprintf(temp, "%d", i);
    strcat(s1, temp);
}
```

În acest program, funcţia **adunsir()** este supraîncărcată. O versiune concatenează două şiruri (cum o face şi **strcat()**). Cealaltă versiune transformă un întreg într-un şir şi apoi îl adaugă altui şir. Aici supraîncărcarea este folosită pentru a crea o interfaţă care adaugă la un şir fie alt şir, fie un întreg.

Puteţi folosi acelaşi nume pentru a supraîncărca funcţii fără legătură, dar nu ar trebui să faceţi aceasta. De exemplu, aţi putea folosi numele **patrat()** pentru a crea funcţii care returnează *pătratul* unui **int** şi *rădăcina pătrată* a unei variabile **double**. Dar aceste două operaţii sunt fundamental diferite; aplicarea supraîncărcării funcţiei în acest mod sfidează scopul iniţial (şi, de fapt, este considerat un stil de programare greşit). În practică trebuie să supraîncărcaţi doar operaţii legate strâns între ele.

Supraîncărcarea operatorilor

Polimorfismul este realizat în C++ şi prin de supraîncărcarea operatorilor. După cum ştiţi, în C++ este posibil să folosiţi operatorii << şi >> pentru a efectua operaţii de I/O de la consolă. Ei pot efectua aceste operaţii suplimentare deoarece operaţiile sunt supraîncărcate în fişierul antet IOSTREAM.H. Când un operator este supraîncărcat, el capătă o semnificaţie suplimentară relativ la o anumită clasă fără a-şi pierde vreunul dintre înţelesurile iniţiale.

În general, puteţi să supraîncărcaţi majoritatea operatorilor din C++ stabilind semnificaţia lor relativ la o anumită clasă. De exemplu, gândiţi-vă la clasa **stiva** prezentată mai devreme, în acest capitol. Este posibil să supraîncărcaţi operatorul + relativ la obiectele de tipul **stiva** astfel încât să adauge conţinutul unei memorii stivă altuia de acelaşi tip. În acelaşi timp, + îşi păstrează semnificaţia sa iniţială relativ la alte tipuri de date.

Deoarece supraîncărcarea operatorilor este în practică ceva mai complexă decât cea a funcţiilor, vom da exemple abia în **Capitolul 14**.

Moştenirea

Cum s-a spus mai devreme, în acest capitol, moştenirea este una dintre caracteristicile cele mai importante ale unui limbaj de programare orientat pe obiecte. În C++, moştenirea este realizată prin acceptarea ca o clasă să încorporeze în declararea sa altă clasă. Moştenirea permite construirea unei ierarhii de clase, trecerea de-la cele mai generale la cele particulare. Procesul implică pentru început definirea *clasei de bază*, care stabileşte calităţile comune ale tuturor obiectelor ce vor deriva din bază. Clasa de bază reprezintă cea mai generală descriere. Clasele derivate din bază se numesc de obicei *clase derivate*. O clasă derivată include toate caracteristicile clasei de bază şi, în plus, calităţi

proprii acelei clase. Pentru a demonstra cum decurg lucrurile, următorul exemplu creează clase care definesc diferite tipuri de clădiri.

Pentru început, este declarată, după cum se arată, clasa **cladire**. Ea va servi ca bază pentru două clase derivate.

```
class cladire {
    int camere;
    int etaje;
    int supraf;
public:
    void nr_camere(int num);
    int cate_camere();
    void nr_etaje(int num);
    int cate_etaje()
    void nr_supraf(int num);
    int cate_supraf();
};
```

Deoarece (în acest exemplu) toate clădirile au trei caracteristici comune - una sau mai multe camere, unul sau mai multe etaje şi o suprafaţă totală, clasa **cladire** include aceste componente în declararea sa. Funcţiile membre care încep cu **nr** iniţializează valorile datelor particulare. Cele care încep cu **cite** returnează aceste valori.

Acum puteţi folosi această definiţie largă pentru a crea clase derivate care descriu anumite tipuri de clădiri. Iată, de exemplu, o clasă derivată numită **casa**:

```
// casa este derivata din cladire
class casa : public cladire {
    int dormitoare;
    int bai;
public:
    void nr_dormitoare(int num);
    int cate_dormitoare();
    void nr_bai(int num);
    int cate_bai();
};
```

Observaţi cum este moştenită **cladire**. Forma generală a moştenirii este:

class *nume_nou_clasa : acces clasa-mostenita {*
    // corpul noii clase
}

Acces este aici opţional. Totuşi, dacă este prezent, el trebuie să fie **public**, **private** sau **protected**. (Aceste opţiuni vor fi studiate mai departe, în Capitolul 12.) Deocamdată, toate clasele moştenite vor fi **public**. Utilizarea modului **public** înseamnă că toate elementele publice ale clasei de bază vor fi publice, de asemenea, şi în clasa derivată care o moşteneşte. De aceea, în acest exemplu, membrii clasei **casa** au acces la funcţiile membre din **cladire** ca şi cum ar fi fost declarate în interiorul lui **casa**. Totuşi, funcţiile membre din **casa** *nu* au acces la părţile particulare din **cladire**. În acest fel, moştenirea nu încalcă principiul încapsulării, necesar în OOP.



***REŢINEŢI:** O clasă derivată are acces atât la proprii membri cât şi la membrii publici ai clasei de bază.*

Iată un program care ilustrează moştenirea. El creează două clase derivate pentru **cladire** folosind moştenirea; unul este **casa**, iar celălalt **scoala**.

```
#include <iostream.h>

class cladire {
    int camere;
    int etaje;
    int supraf;
public:
    void nr_camere(int num);
    int cate_camere();
    void nr_etaje(int num);
    int cate_etaje()
    void nr_supraf(int num);
    int cate_supraf();
};

// casa este derivat din cladire
class casa : public cladire {
    int dormitoare;
    int bai;
public:
    void nr_dormitoare(int num);
    int cate_dormitoare();
    void nr_bai(int num);
    int cate_bai();
};

// scoala este de asemenea derivat din cladire
class scoala : public cladire {
    int saliclasa;
    int laborat;
public:
    void nr_saliclasa(int num);
    int cate_saliclasa();
    void nr_laborat(int num);
    int cate_laborat();
};

void cladire:: nr_camere(int num);
{
    camere = num;
}
void cladire::nr_etaje(int num);
{
    etaje = num;
}

void cladire::nr_supraf(int num);
{
    supraf = num;
}

int cladire::cate_camere()
{
    return camere;
}

int cladire::cate_etaje()
{
    return etaje;
}

int cladire::cate_supraf()
{
    return supraf;
}

void casa::nr_dormitoare(int num);
{
    dormitoare = num;
}
```

```
void casa::nr_bai(int num);
{
    bai = num;
}

int casa::cate_dormitoare()
{
    return dormitoare;
}

int casa::cate_bai()
{
    return bai;
}

void scoala::nr_saliclasa(int num)
{
    saliclasa = num;
}

void scoala::nr_laborat(int num)
{
    laborat = num;
}

int scoala::cate_saliclasa()
{
    return saliclasa;
}

int scoala::cate_laborat()
{
    return laborat;
}

main()
{
    casa c;
    scoala s;
    c.nr_camere(12);
    c.nr_etaje(3);
    c.nr_supraf(4500);
    c.nr_dormitoare(5);
```



```
    c.nr_bai(3);

    cout << "casa are " << h.cate_dormitoare();
    cout << " dormitoare\n";
    s.nr_camere(200);
    s.nr_saliclasa(180);
    s.nr_laborat(5);
    s.nr_supraf(25000);
    cout << "scoala are " << s.cate_saliclasa();
    cout << "sali de clasa\n";
    cout << "Aria sa este de " << s.cate_supraf();

    return 0;
}
```

După cum arată acest program, marele avantaj al moştenirii este acela că puteţi crea o clasificare generală care să fie încorporată în unele particulare. În acest fel, fiecare obiect poate să reprezinte exact propria sa clasificare.

În legătură cu C++, pentru a reprezenta relaţiile de moştenire sunt folosiţi, în general, termenii *bază* şi *derivat*. Totuşi, puteţi să mai întâlniţi termenii *părinte* şi *copil*.

În afară de avantajele clasificării ierarhice, moştenirea mai oferă, de asemenea, suport pentru polimorfismul din timpul rulării prin mecanismul funcţiilor **virtuale**. (Pentru detalii căutaţi în **Capitolul 16.**)

Constructori şi destructori

Este un lucru obişnuit pentru unele părţi ale unui obiect să necesite iniţializare înainte de a putea fi folosite. De exemplu, gândiţi-vă la clasa **stiva** prezentată mai devreme în acest capitol. Înainte ca memoria stivă să poată fi folosită, **vis** trebuie să fie iniţializat cu zero. Acest lucru a fost realizat prin utilizarea funcţiei **init()**. Deoarece cerinţele de iniţializare sunt atât de uzuale, C++ permite obiectelor să se iniţializeze singure, atunci când sunt create. Această iniţializare automată este efectuată prin intermediul unei funcţii constructor.

O *funcţie constructor* este o funcţie specială care este membru al unei clase şi are acelaşi nume cu acea clasă. De exemplu, iată cum arată clasa **stiva** transformată pentru a folosi o funcţie constructor pentru iniţializare:

```
// Aceasta creeaza clasa stiva.
class stiva {
    int stiv[SIZE];
    int vis;
public:
```

```
    stiva();    // constructor
    void pune(int i);
    int scoate();
};
```

Reţineţi că funcţia constructor **stiva()** nu are specificat un tip de returnat. În C++, aceste funcţii nu pot să returneze valori şi, de aceea, nu conţin un tip de returnat.

Funcţia **stiva()** are următorul cod:

```
// functia constructor pentru stiva
stiva::stiva()
{
    vis = 0;
    cout << "Stiva initializata\n";
}
```

Reţineţi că mesajul **Stiva initializata** este afişat ca un mod de semnalare a constructorului. În practica curentă, majoritatea funcţiilor constructor nu vor afişa şi nu vor introduce nimic. Ele vor efectua pur şi simplu diverse iniţializări.

Un constructor al unui obiect este apelat automat atunci când este creat acel obiect. Aceasta înseamnă că el este apelat când se execută declararea sa. Este o mare deosebire între o instrucţiune de declarare de tip C şi una de tip C++. În C, declaraţiile de variabile sunt, să spunem aşa, pasive şi rezolvate în majoritate în timpul compilării. Altfel spus, în C, declaraţiile de variabile nu sunt gândite ca fiind instrucţiuni executabile. Dar, în C++, ele sunt instrucţiuni active, care sunt executate, de fapt, în timpul rulării. Un motiv pentru a fi aşa este acela că o declarare a unui obiect poate necesita apelarea unui constructor, aceasta determinând executarea unei funcţii. Chiar dacă această diferenţă pare acum subtilă şi foarte academică, veţi vedea mai departe că ea are implicaţii importante relativ la iniţializarea variabilelor.

Un constructor al unui obiect este apelat o singură dată pentru obiecte globale sau pentru cele locale de tip **static**. Pentru obiecte locale, constructorul este apelat de fiecare dată când este întâlnită declararea acestuia.

Complementul constructorului este *destructorul*. În multe cazuri, un obiect va trebui să efectueze o anumită acţiune sau unele acţiuni atunci când este distrus. Obiectele locale sunt create când se intră în blocul lor şi distruse când blocul este părăsit. Obiectele globale sunt distruse când se termină programul. Când este distrus un obiect, este apelat destructorul lor (dacă există unul). Există multe motive pentru care poate fi necesară o funcţie destructor. De exemplu, un obiect trebuie să cedeze memoria care i-a fost alocată, sau să închidă un fişier pe care l-a deschis. În C++, cea care tratează evenimentele de la dezactivare este funcţia destructor. Ea are acelaşi nume ca şi constructorul, dar precedat de un caracter ~.

De exemplu, iată clasa **stiva** şi funcţiile sale constructor şi destructor. (Reţineţi că **stiva** nu necesită un destructor; cel prezentat aici este doar pentru demonstraţie.)

```
// Aceasta creeaza clasa stiva.
class stiva {
    int stiv[SIZE];
    int vis;
public:
    stiva();    // constructor
    ~stiva();   // destructor
    void pune(int i);
    int scoate();
};

// functia constructor pentru stiva
stiva::stiva()
{
    vis = 0;
    cout << "Stiva initializata\n";

}
// functia destructor pentru stiva
stiva::~stiva()
{
    cout << "Stiva distrusa\n";
}
```

Observaţi că, la fel ca şi funcţiile constructor, funcţiile destructor nu au valori de returnare.

Iată o nouă versiune a programului **stiva**, studiat mai devreme, în acest capitol, modificat acum pentru a arăta cum lucrează funcţiile constructor şi destructor. Notaţi că nu mai este necesar **init()**.

```
#include <iostream.h>

#define SIZE 100
// Aceasta creeaza clasa stiva.
class stiva {
    int stiv[SIZE];
    int vis;
public:
    stiva();    //constructor
    ~stiva();   //destructor
```

```
    void pune(int i);
    int scoate();
};

// functia constructor pentru stiva
stiva::stiva()
{
    vis = 0;
    cout << "Stiva initializata\n";

}
// functia destructor pentru stiva
stiva::~stiva()
{
    cout << "Stiva distrusa\n";
}

void stiva::pune(int i)
{
    if(vis==SIZE) {
          cout << "Stiva este plina. ";
          return;
    }
    stiv[vis] = i;
    vis++;
}

int stiva::scoate()
{
    if(vis==0) {
          cout << "Depasire inferioara.";
          return 0;
    }
    vis--;
    return stiv[vis];
}

main()
{
    stiva a, b; // creeaza doua obiecte de tip stiva

    a.pune(1);
    b.pune(2);
```



```
    a.pune(3);
    b.pune(4);

    cout << a.scoate() << " ";
    cout << a.scoate() << " ";
    cout << a.scoate() << " ";
    cout << a.scoate() << "\n";

    return 0;
}
```

Acest program afişează următoarele:

```
Stiva initializata
Stiva initializata
3 1 4 2
Stiva distrusa
Stiva distrusa
```

Cuvintele cheie în C++

În plus faţă de cele 32 de cuvinte-cheie ale limbajului C, standardul ANSI propus pentru C++ adaugă încă 30. Aceste cuvinte-cheie sunt prezentate în **Tabelul 11-1**.

asm	overload
bool	private
catch	protected
class	public
const_cast	reinterpret_cast
delete	static_cast
dynamic_cast	template
explicit	this
false	throw
friend	true
inline	try
mutable	typeid
namespace	using
new	virtual
operator	wchar_t

Tabelul 11-1 *Cuvinte-cheie în C++*

În momentul scrierii acestei cărți, cuvintele **bool, const_cast, dynamic_cast, explicit, false, mutable, namespace, reinterpret_cast, static_cast, true, typeid, using** şi **wchar_t** sunt în curs de definire de către comitetul de standardizare ANSI C++ şi s-ar putea să nu fie complet implementate de către compilatorul dvs. Aceste cuvinte-cheie nu au făcut parte din specificația inţială pentru C++ creată de Bjarne Stroustrup. Ele au fost adăugate în primul rând pentru a permite limbajului C++ să se adapteze unor situații speciale şi pot fi modificate. De asemenea, cuvântul cheie **overload** este perimat, dar este inclus pentru compatibilitate cu vechile programe în C++. Ar fi bine să verificați manualul utilizatorului pentru compilatorul dvs. pentru a determina cu exactitate care cuvinte-cheie din C++ sunt admise de acesta.

Forma generală a unui program în C++

Chiar dacă stilurile individuale diferă, majoritatea programelor în C++ vor avea această formă generală:

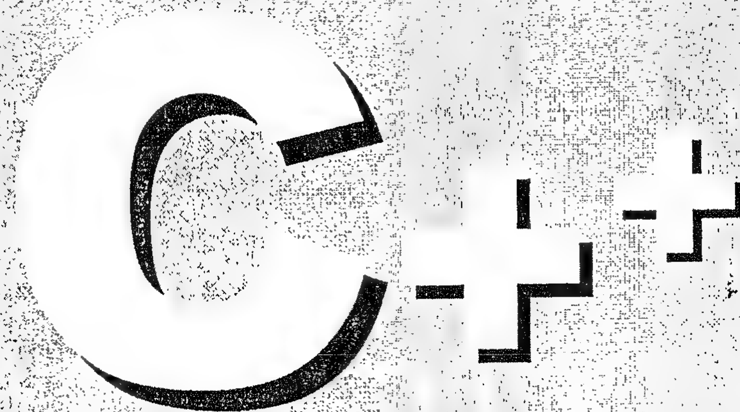
```
#include
declarații clase de bază
declarații clase derivate
prototipuri de funcții nemembre
main()
{
    .
    .
    .
}
definiții de funcții ne-membre
```

Rețineți că, totuşi, în majoritatea proiectelor mari, toate declarațiile pentru **class** vor fi puse într-un fişier antet şi incluse în fiecare modul.

Capitolele care au mai rămas din această secțiune studiază în detaliu atât caracteristicile discutate în acest capitol, cât şi alte facilități ale limbajulului C++.

Capitolul 12

Clase şi obiecte



În C++, clasele formează baza programării orientate pe obiecte. Caracteristic este faptul că o clasă este folosită pentru a defini natura unui obiect. De fapt, în C++ clasa este unitatea de bază pentru încapsulare. În acest capitol sunt examinate în detaliu clasele şi obiectele.

Clase

Clasele sunt create utilizându-se cuvântul cheie **class**. O declarare a unei clase defineşte un nou tip care uneşte cod şi date. Acest nou tip este apoi folosit pentru a declara obiecte din acea clasă. De aceea, o clasă este o abstractizare logică, dar un obiect are o existenţă fizică. Cu alte cuvinte, un obiect este un *exemplar* (o instanţă) al unei clase.

O declarare de clasă este similară sintactic cu cea a unei structuri. În **Capitolul 11** a fost prezentată o formă simplificată de declarare a unei clase. Iată forma generală completă a unei declaraţii de clasă care nu moşteneşte nici o altă clasă.

```
class nume-clasa {
      date şi funcţii particulare
specificator de acces:
      date şi funcţii
specificator de acces:
      date şi funcţii
.
.
.
specificator de acces:
      date şi funcţii
}     lista de obiecte;
```

Lista de obiecte este opţională. Dacă există, ea declară obiecte din acea clasă. Aici, *specificator de acces* este unul dintre aceste trei cuvinte-cheie din C++:

```
public
private
protected
```

Implicit, funcţiile şi datele declarate într-o clasă sunt proprii acelei clase şi doar membrii săi pot să aibă acces la ele. Totuşi, folosind specificatorul de acces **public**, permiteţi funcţiilor sau datelor să fie accesibile altor secţiuni ale programului dvs. O dată utilizat un specificator de acces, efectul său durează până când ori se întâlneşte altul, ori se ajunge la sfârşitul declaraţiei pentru clasă. Pentru a ne reîntoarce la declararea particulară, puteţi folosi specificatorul de acces **private**. Specificatorul **protected** este necesar când este implicată moştenirea (vezi **Capitolul 15**).

Într-o declarare de clasă puteţi să modificaţi specificatorul de clasă oricât de des doriţi. Aceasta înseamnă că puteţi să treceţi la **public** pentru unele declaraţii şi apoi să reveniţi la **private**. Declaraţia de clasă din următorul exemplu ilustrează această facilitate.


```
#include <iostream.h>
#include <string.h>

class angajat {
    char nume[80];
public:
    void punenume(char *n);
    void ianume(char *n);
private:
    double plata;
public:
    void puneplata(double w);
    double iaplata();
};

void angajat::punenume(char *n)
{
    strcpy(nume, n);
}

void angajat::ianume(char *n)
{
    strcpy(n, nume);
}

void angajat::puneplata(double w)
{
    plata = w;
}

double angajat::iaplata()
{
    return plata;
}

main()
```

```
{
    angajat teo;
    char nume[80];

    teo.punenume("Teo Ionescu");
    teo.puneplata(75000);
    teo.ianume(nume);

    cout <<nume << " are $";
    cout << teo.iaplata() << " pe an. ";
    return 0;
}
```

Aici, **angajat** este o simplă clasă care ar putea fi folosită pentru a memora numele şi salariul angajatului. Notaţi că specificatorul de acces **public** este folosit de două ori.

De fapt, majoritatea programatorilor în C++ vor scrie clasa **angajat** aşa cum se arată în continuare, cu toate elementele particulare grupate împreună şi toate elementele publice grupate separat.

```
class angajat {
    char nume[80];
    double plata;
public:
    void punenume(char *n);
    void ianume(char *n);
    void puneplata(double w);
    double iaplata();
};
```

Chiar dacă puteţi folosi specificatorii de acces într-o declaraţie de clasă oricât de des doriţi, singurul avantaj de a o face este acela de a vedea grupări ale diferitelor părţi ale unei clase, putând face mai uşoară citirea şi înţelegerea programului de către altcineva. Totuşi, pentru compilator, nu are importanţă utilizarea mai multor specificatori de acces identici. De fapt, majoritatea programatorilor găsesc că este mai uşor să aibă în fiecare clasă doar câte o singură secţiune **private**, **protected** şi **public**.

Funcţiile care sunt declarate într-o clasă sunt numite *funcţii membre*. Ele pot să aibă acces la orice element al clasei din care fac parte. Aceasta include toate elementele de tip **private**. Variabilele care sunt membre ale unei clase sunt numite *variabile membre* sau *date membre*. În general, orice element al unei clase este numit membru al acelei clase.

Există câteva restricţii care se aplică membrilor clasei. O variabilă membru care nu este de tip **static** nu poate să aibă o iniţializare. Nici un membru nu poate fi un obiect al clasei care se declară. (Deşi un membru poate fi un pointer către clasa care este declarată.) Nici un membru nu poate fi declarat ca **auto**, **extern** sau **register**.

În general, trebuie să faceţi membrii unei clase să fie particulari acelei clase. Aceasta este o parte a modului în care se realizează încapsularea. Totuşi, pot exista situaţii în care veţi avea nevoie să faceţi una sau mai multe variabile publice. (De exemplu, poate fi necesar ca o variabilă folosită intens să fie accesibilă global, pentru a obţine timpi de rulare mai mici.) Când o variabilă este **public**, ea poate fi accesată direct de oricare secţiune a programului. Sintaxa pentru a avea acces la o dată membru **public** este aceeaşi ca pentru apelarea unei funcţii: specificaţi numele obiectului, operatorul punct şi numele variabilei. Următorul program simplu ilustrează accesul direct la variabilele **public**.

```
#include <iostream.h>

class clasamea {
public:
    int i, j, k; // accesibile intregului program
};

main()
{
    clasamea a, b;
    a.i = 100;  //accesul direct la i, j si k
    a.j = 4;
    a.k = a.i * a.j;

    b.k = 12;   // retineti ca a.k si b.k sunt diferite
    cout << a.k << " " << b.k;
    return 0;
}
```

Structuri şi clase

C++ a crescut rolul structurii standard din C la acela al unui mod alternativ, de specificare a unei clase. De fapt, singura diferenţă dintre o clasă şi o **struct** este aceea că, implicit, toţi membrii unei structuri sunt publici, iar cei ai unei clase sunt particulari. În toate celelalte privinţe structurile şi clasele sunt echivalente. Aceasta înseamnă că în C++ o structură defineşte, de asemenea, un tip de clasă. De

exemplu, să considerăm următorul scurt program, care foloseşte o structură pentru a declara o clasă ce controlează accesul la un şir.

```
#include <iostream.h>
#include <string.h>

struct sirulmeu {
    void facesir(char *s); // public
    void aratasir();
private: // acum particulare
    char sir[255];
};

void sirulmeu::facesir(char *s)
{
    if(!*s) *sir = '\0'; // initializeaza sir
    else strcat(sir, s);
}

void sirul meu::aratasir()
{
    cout << sir << "\n";
}
main()
{
    sirulmeu s;

    s.facesir(""); //init
    s.facesir("Va ");
    s.facesir("salut! ");

    s.aratasir();

    return 0;
}
```

Acest program afişează şirul **Va salut!**.

Reţineţi că **s** este declarat în **main()**. Deoarece o **struct** în C++ defineşte un tip de clasă, obiectele de acel tip pot fi declarate folosind doar numele generic al structurii, fără să fie precedat de cuvântul cheie **struct**. Amintiţi-vă că în C variabilele de tip structură trebuie declarate folosind această formă:

struct *nume-generic variabilă*;

Dar, în C++ o structură este un tip de clasă iar numele ei generic descrie un tip complet. Astfel, când declaraţi obiecte nu este nevoie să le precedaţi cu **struct** (însă nici nu este, practic, o eroare dacă faceţi astfel.)

Clasa **sirulmeu** poate fi rescrisă utilizând **class** aşa cum se arată aici:

```
class sirulmeu {
    char sir[255];
public:
    void facesir(char *s); // public
    void aratasir();
};
```

Vă puteţi întreba de ce C++ conţine două cuvinte-cheie practic echivalente, **struct** şi **class**. Această aparentă tautologie este justificată din mai multe motive. În primul rând, nu există nici un motiv fundamental pentru a nu mări capacităţile unei structuri. În C, structurile asigură deja un mod de a grupa date. De aceea, rămâne doar un pas mic să le permitem să accepte funcţii membre. În al doilea rând, deoarece structurile şi clasele sunt înrudite, este mai uşor să translatăm programele existente din C în C++. În sfârşit, chiar dacă astăzi ele sunt echivalente, asigurând două cuvinte-cheie diferite, permitem definiţiei lui **class** să evolueze liber. În schimb, în principiu, pentru ca C++ să rămână compatibil cu C, o declarare a unei structuri poate să nu fie capabilă să evolueze în acelaşi fel.

Chiar dacă puteţi folosi o **struct** acolo unde puteţi folosi o clasă, în general nu ar trebui să o faceţi. Pentru a fi mai clar, ar trebui să utilizaţi o **class** când doriţi o clasă şi o **struct** când doriţi o structură de tip C. Acesta este stilul în care va continua această carte.

 ***REŢINEŢI:** În C++, o declarare a unei structuri defineşte un tip de clasă.*

Uniuni şi clase

Ca şi o structură, o **union** poate fi folosită, de asemenea, pentru a defini o clasă. În C++, uniunile pot conţine atât funcţii membre, cât şi variabile membre. Ele pot să mai includă şi funcţii constructor şi destructor. În C++ o **union** păstrează toate capacităţile din C, cea mai importantă fiind cea prin care toate elementele de date împart aceeaşi locaţie de memorie. Ca şi structura, membrii unei **union** sunt implicit publici. În următorul exemplu este folosit o **union** pentru a inversa între ei octeţii care formează un întreg de tip **unsigned**. (Acest exemplu presupune că întregii au lungimea de doi octeţi.)

```
#include <iostream.h>

union sch_octet {
    void sch();
    void da_octet(unsigned i);
    void arata_cuvant();

    unsigned u;
    unsigned char c[2];
};
void sch_octet::sch()
{
    unsigned char t;

    t = c[0];
    c[0] = c[1];
    c[1] = t;
}

void sch_octet::arata_cuvant()
{
    cout << u;
}

void sch_octet::da_octet(unsigned i)
{
    u = i;
}

main()
{
    sch_octet b;

    b.da_octet(49034);
    b.sch();
    b.arata_cuvant();

    return 0;
}
```

Este important să înțelegeți că, la fel ca și o stuctură, o declarație a unei **union** în C++ definește un tip special de clasă. Aceasta înseamnă că se păstrează principiul încapsulării.

Există mai multe restricții care trebuie cunoscute atunci când folosiți uniunile în C++. În primul rând, o **union** nu poate moșteni nici un alt tip de clasă. Mai mult, o **union** nu poate fi o clasă de bază. O **union** nu poate avea funcții virtuale membre. (Funcțiile virtuale vor fi discutate în **Capitolul 16.**) Variabilele de tip **static** nu pot fi membri ai unei **union.** O **union** nu poate avea ca membru nici un obiect care are supraîncărcat operatorul =. În sfârșit, nici un obiect nu poate fi membru al unei uniuni dacă acel obiect conține o funcție constructor sau destructor.

Uniuni anonime

În C++ există un tip special de **union** numit *uniune anonimă*. O uniune anonimă nu conține un nume de tip și nici o variabilă nu poate fi declarată ca fiind de acel tip de uniune. În schimb, ea spune compilatorului că variabilele membre ale uniunii vor împărți aceeași locație. Dar, variabilele însele sunt utilizate direct, fără sintaxa normală cu operator punct. Să luăm, de exemplu, acest program:

```
#include <iostream.h>
#include <string.h>

main()
{
    // definirea uniunii anonime
    union {
          long l;
          double d;
          char s[4];
    };

    // acum, accesul la membrii uniunii este direct
    l = 100000;
    cout << l << " ";
    d = 123.2342;
    cout << d << " ";
    strcpy(s, "hi");
    cout <<s;

 return 0;
 }
```

După cum puteți vedea, elementele din **union** sunt utilizate ca și cum ar fi fost declarate ca variabile locale normale. De fapt, exact așa le veți și folosi în programul dvs. Mai mult, chiar dacă ele sunt definite într-o declarație de tip **union**, ele se află la același nivel al sferei de acțiune ca și oricare alte variabile locale din același bloc. Într-adevăr, membrii unei uniuni anonime nu pot să aibă același nume ca al nici unui alt identificator cunoscut în domeniul curent. Aceasta înseamnă că

numele membrilor unei uniuni anonime nu trebuie să intre în conflict cu alți identificatori cunoscuți din domeniul uniunii.

Toate restricțiile pentru o **union** se aplică și celor anonime, cu câteva completări. Mai întâi, singurele elemente care pot fi conținute într-o uniune trebuie să fie de tip date. Nu sunt permise funcții membre. Uniunile anonime nu pot conține elemente de tip **private** sau **protected**. În sfârșit, uniunile globale anonime trebuie specificate ca fiind de tip **static**.

Funcții prietene

Este posibil să permiteți unei funcții care nu este membru să aibă acces la membrii particulari ai clasei folosind un **friend** (prieten). O funcție **friend** are acces la membrii **private** și **protected** ai clasei căreia îi este prietenă. Pentru a declara o funcție **friend**, includeți prototipul ei în acea clasă, precedat de cuvântul cheie **friend**. Să luăm acest program:

```
#include <iostream.h>

class clasamea {
   int a, b;
public:
   friend int sum(clasamea x);
   void da_ab(int i, int j);
};

void clasamea::da_ab(int i, int j)
{
   a = i;
   b = j;
}

// Nota: sum() nu este o functie membra a nici unei clase.
int sum(clasamea x)
{
   /* Deoarece sum() este functie prietena pentru
      clasamea, ea poate avea acces direct la a si b. */
   return x.a + x.b;
}

main()
{
   clasamea n;
   n.da_ab(3, 4);
```



```
   cout << sum(n);
   return 0;
}
```

În acest exemplu, funcția **sum()** nu este un membru din **clasamea**. Totuși, ea are acces deplin la membrii particulari. De asemenea, rețineți că **sum()** este apelată normal. Deoarece ea nu este o funcție membră a clasei, ea nu trebuie (și nici nu poate) să aibă un nume de obiect.

Chiar dacă nu s-a câștigat nimic făcând **sum()** să fie **friend** în loc de a fi un membru tip funcție al clasei **clasamea**, există anumite condiții în care funcțiile **friend** sunt într-adevăr valoroase. În primul rând, prietenii pot să fie folositori când supraîncărcați anumite tipuri de operatori (a se vedea **Capitolul 14**). În al doilea rând, funcțiile **friend** fac mai ușoară crearea anumitor tipuri de funcții de I/O (a se vedea Capitolul 17). Al treilea motiv pentru care sunt folositoare funcțiile **friend** este că, în unele cazuri, două sau mai multe clase pot conține membri care sunt corelați cu alte secțiuni ale programului. Să examinăm acum acest al treilea mod de folosire.

Pentru început imaginați-vă două clase diferite, fiecare dintre ele afișând pe ecran un mesaj când apar erori. Alte secțiuni ale programului pot să dorească să știe dacă mesajul a fost afișat înainte de a fi fost scris pe ecran, astfel încât să nu se suprapună, din greșeală, alt mesaj. Chiar dacă puteți crea funcții membre în fiecare clasă care returnează o valoare ce indică dacă un mesaj este activ, aceasta înseamnă un efort în plus la verificarea condiției (două apelări ale funcției în loc de una). Dacă acea condiție trebuie verificată des, munca în plus poate să nu fie acceptabilă. Dar, utilizând pentru fiecare clasă o funcție **friend**, este posibil să verificați starea fiecărui obiect apelând doar această singură funcție. De aceea, în asemenea situații, o funcție **friend** permite să creați un cod mai eficient. Următorul program ilustrează acest concept:

```
#include <iostream.h>

#define IDLE 0
#define INUSE 1

class C2; // explicata mai jos
class C1 {
   int stare;  // INUSE daca este pe ecran, IDLE daca nu
   // ...
public:
   void da_stare(int cond);
   friend int idle(C1 a, C2 b);
};
```

```
class C2 {
    int stare; // INUSE daca este pe ecran, IDLE daca nu
    // ...
public:
    void da_stare(int cond);
    friend int idle(C1 a, C2 b);
};

void C1::da_stare(int cond)
{
    stare = cond;
}

void C2::da_stare(int cond)
{
    stare = cond;
}

int idle(C1 a, C2 b)
{
    if(a.stare || b.stare) return 0;
    else return 1;
}

main()
{
    C1 x;
    C2 y;

    x.da_stare(IDLE);
    y.da_stare(IDLE);

    if(idle(x, y)) cout << "Ecranul poate fi folosit.\n";
    else cout << "In curs de folosire.\n";
    x.da_stare(INUSE);
    if(idle(x, y)) cout << "Ecranul poate fi folosit.\n";
    else cout << "In curs de folosire.\n";

    return 0;
}
```

Rețineți că acest program folosește pentru clasa **C2** o *declarare ulterioară* (numită și *referire ulterioară*). Acest lucru este necesar deoarece declararea

funcției **idle()** în interiorul lui **C1** se referă la **C2** înainte de a fi declarată. Pentru a crea o declarare ulterioară a unei clase, folosiți pur și simplu forma prezentată în acest program.

Un **friend** al unei clase poate fi membru al altei clase. De exemplu, iată programul anterior rescris astfel încât **idle()** este un membru al lui **C1**:

```
#include <iostream.h>

#define IDLE 0
#define INUSE 1

class C2; // explicata mai jos

class C1 {
    int stare; // INUSE daca este pe ecran, IDLE daca nu
    // ...
public:
    void da_stare(int cond);
    int idle(C2 b); // acum un membru al lui C1
};

class C2 {
    int stare; // INUSE daca este pe ecran, IDLE daca nu
    // ...
public:
    void da_stare(int cond);
    friend int C1::idle(C2 b);
};

void C1::da_stare(int cond)
{
    stare = cond;
}

void C2::da_stare(int cond)
{
    stare = cond;
}

// idle() este un membru al lui C1, dar prieten al lui C2
int C1::idle(C2 b)
{
    if(stare || b.stare) return 0;
```

```
    else return 1;
}

main()
{
    C1 x;
    C2 y;

    x.da_stare(IDLE);
    y.da_stare(IDLE);

    if(x.idle(y)) cout << "Ecranul poate fi folosit.\n";
    else cout << "In curs de folosire.\n";

    x.da_stare(INUSE);

    if(x.idle(y)) cout << "Ecranul poate fi folosit.\n";
    else cout << "In curs de folosire.\n";

    return 0;
}
```

Deoarece **idle()** este membru al lui **C1**, ea poate avea acces direct la variabila **stare** din obiectele de tip **C1**. De aceea, doar obiectele de tip **C2** trebuie să îi fie transmise.

Există două restricţii importante care se aplică funcţiilor **friend**. Prima, o clasă derivată nu poate moşteni funcţii **friend**. A doua, o funcţie **friend** nu poate avea un specificator de clasă de memorare. Aceasta înseamnă că ea nu poate fi declarată ca fiind **static** sau **extern**.

Clase prietene

Este posibil ca o clasă să fie **friend** pentru alta. Când este aşa, clasa **friend** are acces la numele particulare definite în cadrul celeilalte. Aceste nume particulare pot include lucruri ca nume de tipuri şi enumerări de constante. Să luăm un exemplu:

```
#include <iostream.h>

class monede {
    // Urmatoarea este o enumerare particulara.
    enum unitati {penny, nickel, dime, quarter,
                  jumatate_dolar};
    friend class cantitate;
};

class cantitate {
    monede::unitati bani; // retineti modul monede::unitati
public:
    void dam();
    int iam();
} ob;

void cantitate::dam()
{
    // Unitatile de enumerare sunt accesibile aici deoarece
    // cantitate este friend pentru monede.
    bani = monede::dime;
}

int cantitate::iam()
{
    return bani;
}

    main()
{
    ob.dam();

    cout << ob.iam(); // afiseza numarul 2

    return 0;
}
```

Aici, clasa **cantitate** are acces la specificatorul de tip **unitati** declarat în clasa **monede** (şi la numele definite în enumerarea **unitati**) deoarece **cantitate** este un friend pentru **monede**.

Este important să înţelegeţi că atunci când o clasă este **friend** pentru o alta, ea are acces doar la numele definite în interiorul celeilalte. Ea nu moşteneşte cealaltă clasă. Mai precis, membrii primei clase nu devin membri ai clasei **friend**.

Clasele prietene sunt folosite deseori. Ele permit tratarea anumitor situaţii speciale.

Funcţii inline

Există o caracteristică importantă în C++, numită *funcţie inline*, care este folosită uzual împreună cu clasele. Funcţiile inline sunt studiate aici, deoarece restul

capitolului (şi restul cărţii) le va utiliza intens.

În C++ puteţi crea funcţii scurte care nu sunt apelate efectiv; în loc de aceasta codul lor se dezvoltă în interior, în fiecare punct în care sunt numite. Acest proces este similar cu folosirea funcţiilor macro. Pentru a determina ca o funcţie să se dezvolte inline în loc să fie apelată, precedaţi definirea ei cu cuvântul cheie **inline**. De exemplu, în programul următor, funcţia **max()** este dezvoltată inline în loc să fie apelată.

```
#include <iostream.h>

inline int max(int a, int b)
{
    return a>b ? a : b;
}

main()
{
    cout << max(10, 20);
    cout << " " << max(99, 88);
    return 0;
}
```

În ceea ce priveşte compilatorul, programul precedent este echivalent cu acesta:

```
#include <iostream.h>

main()
{
    cout << (10>20 ? 10 : 20);
    cout << " " << (99>88 ? 99 : 88);

    return 0;
}
```

Motivul pentru care funcţiile **inline** sunt o facilitate în plus în C++ este că ele permit să creaţi coduri foarte eficiente. De vreme ce pentru clase este tipic ca, deseori, să solicite executarea frecventă a funcţiilor de interfaţă (care asigură accesul la datele particulare), eficienţa acestor funcţii este o cerinţă esenţială în C++. După cum ştiţi probabil, la fiecare apelare a unei funcţii se generează o cantitate de muncă suplimentară prin mecanismul de apelare şi returnare. În mod normal, când este apelată o funcţie, argumentele sunt puse în memoria stivă şi sunt salvate mai multe registre şi apoi rememorate când funcţia se returnează.

Problema este că aceste instrucţiuni iau timp. Dar, când o funcţie se dezvoltă inline, nu mai apare nici una dintre aceste operaţii. Chiar dacă dezvoltarea inline a funcţiilor poate determina timpi de rulare mai scurţi, deseori rezultă dimensiuni mai mari de coduri datorită instrucţiunilor duplicate. Din acest motiv, este bine să introducem inline doar funcţii foarte mici. Mai mult, o idee bună este, de asemenea, să dezvoltaţi inline doar funcţiile care vor avea un impact mare asupra performanţelor programului dvs.

Ca şi specificatorul **register**, **inline** este, practic, pentru compilator doar o *solicitare*, nu o comandă. Compilatorul poate să aleagă ignorarea ei. De asemenea, unele compilatoare nu pot introduce inline toate tipurile de funcţii. De exemplu, în general, un compilator nu o face pentru o funcţie recursivă. Va trebui să controlaţi în manualul compilatorului dvs. restricţiile pentru **inline**. Amintiţi-vă că dacă o funcţie nu poate fi introdusă **inline**, ea trebuie apelată ca o funcţie normală.

Funcţiile inline pot fi membre ale unei clase. De exemplu, acesta este un program perfect valabil în C++:

```
#include <iostream.h>

class clasamea {
    int a, b;
public:
    void init(int i, int j);
    void arata();
};

inline void clasamea::init(int i, int j)
{
    a = i;
    b = j;
}

inline void clasamea::arata()
{
    cout << a << " " << b << "\n";
}

main()
{
    clasamea x;
    x.init(10, 20);
    x.arata();
```

```
    return 0;
}
```

Definirea funcţiilor inline într-o clasă

Este posibil să definiţi funcţii scurte într-o declarare de clasă. Când o funcţie este definită într-o declarare a unei clase, ea este automat transformată într-o funcţie **inline** (dacă este posibil). Nu este necesar (dar nici greşit) să precedaţi declararea sa cu cuvântul cheie **inline**. De exemplu, programul precedent este rescris aici cu definirile pentru **init()** şi **arata()** incluse în declararea lui **clasamea**.

```
#include <iostream.h>

class clasamea {
    int a, b;
public:
    // inline automat
    void init(int i, int j)  {a=i; b=j;}
    void arata()  {cout << << " " << b << "\n";}
};

main()
{
    clasamea x;
    x.init(10, 20);
    x.arata();

    return 0;
}
```

Reţineţi forma codului funcţiei din **clasamea**. Deoarece funcţiile **inline** sunt, de obicei, scurte, acest stil de codare din cadrul unei clase este aproape tipic. Totuşi, sunteţi liber să folosiţi orice formă doriţi. De exemplu, iată o cale perfect valabilă de rescriere a declarării pentru **class**:

```
#include <iostream.h>

class clasamea {
    int a, b;
public:
    // inline automat
    void init(int i, int j)
    {
    a = i;
    b = j;
    }

    void arata(){
    cout << << " " << b << "\n";
    }
};
```

Practic, dezvoltarea inline a funcţiei **arata()** este fără sens deoarece (în general) timpul luat de instrucţiunea de I/O va depăşi mult pe cel al apelării unei funcţii. Totuşi, este foarte uzual în C++ ca toate funcţiile membre să fie definite în interiorul clasei lor. (De fapt, în programele profesionale scrise în cod C++ funcţiile membre sunt definite foarte rar în afara declarării clasei lor.)

Reţineţi că funcţiile constructor şi destructor pot fi, de asemenea, dezvoltate inline - ori implicit, dacă sunt definite în clasa lor, ori explicit.

Funcţii constructor cu parametri

Este posibil să transmiteţi argumente către funcţiile constructor. Tipic, aceste argumente sunt folosite pentru a contribui la iniţializarea unui obiect atunci când este creat. Pentru a crea un constructor cu parametri, îi adăugaţi simplu parametri în modul în care aţi face-o pentru orice altă funcţie. Funcţia este în aşa fel creată încât parametrii sunt folosiţi pentru a iniţializa obiectul. Iată, de exemplu, o **class** simplă care include constructori cu parametri:

```
#include <iostream.h>

class clasamea {
    int a, b;
public:
    clasamea(int i, int j) {a=i; b=j;}
    void arata() {cout << a << " " << b;}
};

main()
{
    clasamea ob(3, 5);
    ob.arata();

    return 0;
}
```

Rețineți că în definirea lui **clasamea()**, parametrii **i** și **j** sunt folosiți pentru a da valori inițiale în **a** și **b**.

Programul ilustrează cea mai uzuală cale de a specifica argumente atunci când declarați un obiect care folosește o funcție constructor cu parametri. Instrucțiunea:

```
clasamea ob(3, 4);
```

determină crearea unui obiect numit **ob** și se pasează argumentele **3** și **4** către parametrii **i** și **j** din **clasamea**. Puteți să pasați argumente folosind și acest tip de instrucțiune de declarare:

```
clasamea ob = clasamea(3, 4);
```

Totuși, prima metodă este una general utilizată, iar ea va fi utilizată în majoritatea exemplelor din această carte. De fapt, există practic o mică diferență între cele două tipuri de declarații care se referă la copierea funcției constructor. (Copierea constructorilor este discutată în Capitolul 22.)

Iată un alt exemplu care folosește o funcție constructor cu parametri. Ea creează o **class** care păstrează informații despre cărțile din bibliotecă.

```
#include <iostream.h>
#include <string.h>

#define IN 1
#define CHECKED_OUT 0

class carte {
   char autor[40];
   char titlu[40];
   int stare;
public:
   carte(char *n, char *t, int s);
   int ia_stare() {return stare;}
   void da_stare(int s) {stare = s;}
   void arata();
};

carte::carte(char *n, char *t, int s)
{
   strcpy(autor, n);
   strcpy(titlu, t);
   stare = s;
}
```



```
void carte::arata()
{
   cout << titlu << " de " <<autor;
   cout << " este ";
   if(stare==IN) cout << "aici.\n";
   else cout << "data.\n";
}

main()
{
   carte b1("Twain", "Tom Sawyer", IN);
   carte b2("Melville", "Moby Dick", CHECKED_OUT);

   b1.arata();
   b2.arata();

   return 0;
}
```

Funcțiile constructor cu parametri sunt foarte folositoare deoarece ele permit să evitați să faceți o apelare în plus a funcției doar pentru a inițializa una sau mai multe variabile dintr-un obiect. Fiecare apelare a funcției pe care o puteți evita face programul mai eficient. De asemenea, rețineți că funcțiile **da_stare()** și **ia_stare()** sunt definite în interiorul clasei **carte**. Aceasta este o practică foarte uzuală când scrieți programe în C++.

Funcțiile constructor cu un parametru: un caz special

Dacă o funcție constructor are doar un parametru, există o a treia cale de a-i pasa o valoare inițială. Să luăm, de exemplu, următorul program scurt:

```
#include <iostream.h>

class X {
   int a;
public:
   X(int j) { a = j; }
   int daa() { return a; }
};

main()
{
```

```
    X ob = 99;  // paseaza 99 lui j
    cout << ob.daa(); // afiseaza 99
    return 0;
}
```

Aşa cum arată exemplul acesta, în cazul în care constructorul are doar un argument, puteţi folosi pur şi simplu forma de iniţializare normală. Compilatorul de C++ va atribui automat valoarea din dreapta semnului = parametrului constructorului.

Membrii de tip static ai claselor

Atât funcţiile membre cât şi datele membre ale unei clase pot fi declarate **static**. Acest paragraf explică ce înseamnă aceasta, relativ la fiecare tip de membru.

Membri statici de tip date

Când precedaţi o declaraţie a unei variabile membru cu **static**, îi spuneţi compilatorului că va exista doar o copie a acelei variabile şi va fi folosită de către toate obiectele clasei. Spre deosebire de membrii tip date obişnuiţi, nu sunt create copii individuale ale variabilelor membre **static** pentru fiecare obiect. Nu are importanţă câte obiecte de acel tip de clasă sunt create, va exista doar o singură copie a membrilor date de tip **static**. De aceea, toate obiectele acelei clase folosesc aceeaşi variabilă. Când este creat primul obiect, toate variabilele de tip **static** sunt iniţializate cu zero.

Când declaraţi o dată membru ca fiind **static** într-o clasă, *nu* o definiţi. Cu alte cuvinte, nu îi alocaţi memorie. (În limbajul C++, o declaraţie *descrie* ceva. O definire face ca ceva să existe.) În schimb, trebuie să daţi o definire globală a membrilor de tip date **static** undeva, în afara clasei. Aceasta se face redeclarând variabila **static** folosind operatorul de specificare a domeniului pentru a preciza clasa căreia îi aparţine; ca urmare se va aloca memorie pentru variabilă. (Amintiţi-vă că o declarare de **class** este o simplă construcţie logică ce nu are suport material.)

Pentru a înţelege utilizarea şi efectul unui membru de tip date **static**, să luăm acest program:

```
#include <iostream.h>

class comun {
    static int a;
    int b;
public:
    void da(int i, int j) {a=i; b=j;}
```



```
    void arata();
};

int comun::a; // defineste a
void comun::arata()
{
    cout << "Acesta este a static: " << a;
    cout << "\nAcesta este b ne-static: " << b;
    cout << "\n";
}

main()
{
    comun x, y;

    x.da(1, 1); // initializeaza pe a cu 1
    x.arata();

    y.da(2, 2); // schimba pe a in 2
    y.arata();

    x.arata(); /* Aici a a fost schimbat atat pentru x cat
                  si pentru y deoarece a este comun pentru
                  ambele obiecte. */

    return 0;
}
```

Când este rulat acest program afişează următoarea ieşire:

```
Acesta este a static: 1
Acesta este b ne-static: 1
Acesta este a static: 2
Acesta este b ne-static: 2
Acesta este a static: 2
Acesta este b ne-static: 1
```

Reţineţi că numărul întreg a este declarat atât în **comun** cât şi în afara sa. Cum am spus mai devreme, acest lucru este necesar deoarece declararea lui **a** în interiorul lui **comun** nu determină alocare de memorie.

***NOTĂ:** Prin convenţie, versiunile vechi de C++ nu impun a doua declaraţie a unei variabile membru **static**. Dar această convenţie a dat naştere la numeroase inconveniente grave şi a fost eliminată acum câţiva ani. Chiar şi aşa, puteţi încă să mai găsiţi coduri vechi de C++ care nu redeclară variabilele membre **static**. În aceste cazuri, va fi necesar să adăugaţi definirile cerute.*

O variabilă membru **static** există *înainte* de a fi creat orice obiect din acea clasă. De exemplu, în următorul scurt program, **a** este atât **public** cât şi **static**. De aceea, el poate fi utilizat direct în **main()**. Mai mult, deoarece **a** există înainte de crearea unui obiect din **comun**, lui **a** i se poate da oricând o valoare. Aşa cum ilustrează programul, valoarea lui **a** nu este modificată de crearea obiectului **x**. Din acest motiv, amândouă instrucţiunile de ieşire afişează aceeaşi valoare: 99.

```
#include <iostream.h>

class comun {
public:
    static int a;
};

int comun::a; // defineste pe a

main()
{
    // initializeaza pe a inaintea crearii oricaror obiecte
    comun::a = 99;

    cout << "Aceasta este valoarea initiala a lui a: "
         << comun::a;
    cout << "\n";
    comun x;
    cout << "Acesta este x.a: " << x.a;

    return 0;
}
```

Reţineţi cum se face accesul la **a** prin folosirea numelui clasei şi a operatorului de specificare a domeniului. În general, când programul dvs. are acces la un membru **static** independent de un obiect, trebuie să îl precizaţi folosind numele clasei al cărei membru este.

Una dintre cele mai obişnuite utilizări ale variabilelor membre **static** este de a asigura controlul accesului la unele resurse comune. De exemplu, puteţi crea mai multe obiecte, fiecare dintre ele trebuind să scrie într-un anumit fişier de pe disc. Este evident, totuşi, că doar unui singur obiect îi este permis să scrie într-un fişier într-un anumit moment. În acest caz, este bine să declaraţi o variabilă **static** care să indice când este folosit fişierul şi când este liber. Fiecare obiect va controla apoi această variabilă înainte de a scrie în fişier. Următorul program arată cum puteţi folosi variabila **static** de acest tip pentru a controla accesul la o resursă limitată.


```
#include <iostream.h>

class cl {
    static int resursa;
public:
    int ia_resursa();
    void resursa_libera() {resursa = 0;}
};

int cl::resursa; // defineste resursa
int cl::ia_resursa()
{
    if(resursa) return o; // resursa este in lucru
    else {
          resursa = 1;
          return 1; // resursa atribuita acestui obiect
    }
}

main()
{
    cl ob1, ob2;

    if(ob1.ia_resursa()) cout << "ob1 are dreptul
                                 la resursa\n";

    if(!ob2.ia_resursa()) cout << "ob2 nu are dreptul
                                  la resursa\n";

    ob1.resursa_libera(); // o lasa pentru altceva

    if(ob2.ia_resursa())
          cout << "ob2 poate acum sa foloseasca resursa\n";

    return 0;
}
```

Folosind variabilele membre **static**, virtual ar trebui să nu mai aveţi nevoie de nici o variabilă globală. Problema variabilelor globale pentru OOP este aceea că ele încalcă, aproape întotdeauna, principiul încapsulării.

Funcţii membre statice

Funcţiile membre pot fi, de asemenea, declarate ca **static**. Există mai multe restricţii relativ la acestea. În primul rând, ele pot să aibă acces doar la alţi membri

de tip **static** ai clasei, şi, bineînţeles, la funcţiile şi datele globale. În al doilea rând, ele nu pot avea un pointer de tip **this**. (Pentru informaţii despre **this** citiţi **Capitolul 13**.) În al treilea rând, nu poate exista o versiune **static** şi una ne-**static** ale aceleiaşi funcţii.

În continuare este prezentată o versiune uşor modificată a programului care demonstrează secţiunea anterioară. Observaţi că **ia_resursa** este acum declarată **static**. După cum se vede din program, accesul la **ia_resursa** este permis fie funcţiei înseşi, fie independent de vreun obiect utilizând numele clasei şi operatorul de specificare a domeniului, fie în legătură cu un obiect.

```
#include <iostream.h>

class cl {
    static int resursa;
public:
    static int ia_resursa();
    void resursa_libera()  {resursa = 0;}
};
int cl::resursa; // defineste resursa
int cl::ia_resursa()
{
    if(resursa) return 0; // resursa este in lucru
    else {
          resursa = 1;
          return 1; // resursa atribuita acestui obiect
    }
}

main()
{
    cl ob1, ob2;
    /* ia_resursa este static astfel incat poate fi apelata
       independent de orice obiect. */
    if(cl::ia_resursa()) cout << "ob1 are dreptul
                                  la resursa\n";
    if(!cl::ia_resursa()) cout << "ob2 nu are dreptul
                                  la resursa\n";

    ob1.resursa_libera();

    if(ob2.ia_resursa()) // se poate folosi si apelul cu obiect
          cout << "ob2 poate acum sa foloseasca resursa\n";
```



```
    return 0;
}
```

De fapt, funcţiile membre **static** au aplicaţii limitate, dar o bună utilizare a lor este aceea că ele pot „preiniţializa" datele particulare de tip **static**, înainte de crearea efectivă a vreunui obiect. De exemplu, acesta este un program perfect valabil în C++:

```
#include <iostream.h>

class tip_static {
    static int i;
public:
    static void init(int x) {i = x;}
    void arata() {cout << i;}
};

int tip_static::i; // defineste i

main()
{
    // initializeaza date de tip static inainte de crearea obiectului
    tip_static::init(100);

    tip_static x;
    x.arata(); // afiseaza 100

    return 0;
}
```

Când sunt executaţi constructorii şi destructorii

Ca regulă generală, un constructor de obiecte este apelat la declararea obiectului, iar un destructor de obiecte este apelat când este distrus obiectul. Vom prezenta acum momentul exact al apariţiei acestor evenimente.

O funcţie constructor de obiecte este executată când este întâlnită instrucţiunea de declarare a obiectului. Mai mult, când două sau mai multe obiecte sunt declarate în aceeaşi instrucţiune, constructorii sunt apelaţi în ordinea în care sunt ele întâlnite, de la stânga la dreapta. Funcţiile destructor pentru obiecte locale sunt executate în ordine inversă faţă de cele constructor.

Funcţiile constructor pentru obiectele globale sunt executate *înaintea* lui **main()**. Constructorii globali din acelaşi fişier sunt executaţi în ordine, de la stânga la

dreapta şi de sus în jos. Nu puteţi să ştiţi ordinea execuţiei constructorilor globali împrăştiaţi prin mai multe fişiere. Destructorii globali se execută în ordine inversă *după* ce se încheie **main()**.

Următorul program ilustrează executarea constructorilor şi destructorilor.

```
#include <iostream.h>

class clasamea {
public:
    int cine;
    clasamea(int ex);
    ~clasamea();
} glob_ob1(1), glob_ob2(2);

clasamea::clasamea(int ex)
{
    cout << "Initializare " << ex << "\n";
    cine = ex;
}

clasamea::~clasamea()
{
    cout << "Distrugere " << cine << "\n";
}

main()
{
    clasamea local_ob1(3);

    cout << "Aceasta nu va fi prima linie afisata.\n";

    clasamea local_ob2(4);

    return 0;
}
```

El afişează această ieşire:

```
Initializare 1
Initializare 2
Initializare 3

Aceasta nu va fi prima linie afisata.
```



```
Initializare 4
Distrugere 4
Distrugere 3
Distrugere 2
Distrugere 1
```

Operatorul de specificare a domeniului

După cum ştiţi, operatorul :: este folosit la asocierea unui nume de clasă cu un nume de membru pentru a spune compilatorului cărei clase aparţine acel membru. Dar, operatorul de specificare a domeniului are şi o altă utilizare asemănătoare: el poate permite accesul la un nume dintr-un domeniu închis, nume care este „ascuns" de o declarare locală a aceluiaşi nume. Să luăm, de exemplu, acest fragment:

```
.
.
.
int i; // i global

void f()
{
    int i; // i local

    i = 10; // foloseste i local
    .
    .
    .
}
.
.
.
```

Totuşi, ce se întâmplă dacă funcţia f() are nevoie de acces la versiunea globală a lui i? Poate să îl capete dacă i este precedat de operatorul ::, aşa cum se arată aici:

```
.
.
.
int i; // i global
```

```
void f()
{
    int i; // i local
    ::i = 10; // acum se refera la i global
    .
    .
    .
}
.
.
.
```

Clase imbricate

Este posibil să definiţi o clasă în cadrul alteia. Procedând astfel, creaţi clase *imbricate*. Deoarece o declarare a unei **class** defineşte, de fapt, un domeniu de influenţă, o clasă imbricată este validă doar în interiorul clasei ce o conţine. Clasele imbricate sunt folosite rareori. Datorită flexibilităţii şi puterii mecanismului de moştenire, practic nu este nevoie de clase imbricate.

Clase locale

O clasă poate fi definită în interiorul unei funcţii. De exemplu, acesta este un program C++ valid:

```
#include <iostream.h>

void f();

main()
{
    f();
    // clasamea nu este cunoscuta aici
    return 0;
}

void f()
{
    class clasamea {
            int i;
    public:
            void pune_i(int n) {i=n;}
            int da_i() {return i;}
    } ob;

    ob.pune_i(10);
    cout << ob.da_i();
}
```

Când o clasă este declarată într-o funcţie, ea este cunoscută doar acelei funcţii şi necunoscută în afara ei.

Claselor locale li se aplică mai multe restricţii. Prima, toate funcţiile membre trebuie definite în interiorul declaraţiei pentru **class**. Clasa locală nu poate să folosească sau să aibă acces la variabilele locale ale funcţiei în care este declarată (cu excepţia variabilelor locale de tip **static** declarate în interiorul funcţiei). În interiorul unei clase locale nu poate fi declarată nici o variabilă de tip **static**. Din cauza acestor restricţii, clasele locale nu sunt uzuale în programarea în C++.

Transmiterea obiectelor către funcţii

Obiectele pot fi transmise (pasate) către funcţii exact ca oricare alt tip de variabilă. Obiectele sunt pasate funcţiilor prin utilizarea mecanismului standard apelare-prin-valoare, adică prin copiere. Dar efectuarea unei copii înseamnă practic crearea unui alt obiect. Aceasta ne face să ne întrebăm dacă este executată funcţia constructor a obiectului la crearea copiei şi dacă este executată funcţia destructor la distrugerea copiei. Răspunsul la aceste două întrebări vă poate surprinde. Pentru început, iată un exemplu:

```
#include <iostream.h>

class clasamea {
    int i;
public:
    clasamea(int n);
    ~clasamea();
    void pune_i(int n) {i=n;}
    int da_i() {return i;}
};

clasamea::clasamea(int n)
{
    i = n;
    cout << "Construieste " << i << "\n";
}
```

```
clasamea::~clasamea()
{
    cout << "Distruge " << i << "\n";
}

void f(clasamea ob);

main()
{
    clasamea o(1);
    f(o);
    cout << "Acesta este i din main():";
    cout << o.da_i() << "\n";

    return 0;
}

void f(clasamea ob)
{
    ob.pune_i(2);

    cout << "Acesta este i local: " << ob.da_i();
    cout << "\n";
}
```

Acest program are următoarea ieşire:

```
Construieste 1
Acesta este i local: 2
Distruge 2
Acesta este i din main(): 1
Distruge 1
```

Reţineţi că sunt executate două apelări ale funcţiei destructor, dar una singură a funcţiei constructor. După cum ilustrează ieşirea, funcţia constructor nu este apelată când copia lui **o** (din **main()**) este transmisă lui **ob** (din **f()**). Motivul pentru care ea nu este apelată când este făcută copia obiectului este uşor de înţeles. Când transmiteţi un obiect unei funcţii, doriţi starea curentă a obiectului. Dacă funcţia constructor este apelată când este creat obiectul, acesta va fi iniţializat, fiind posibilă modificarea sa. De aceea, funcţia constructor nu poate fi executată atunci când este generată copia unui obiect printr-o apelare de funcţie.

Deşi nu este apelată funcţia constructor când un obiect este pasat unei funcţii, la distrugerea copiei este, totuşi, necesar să apelăm destructorul. (Copia este distrusă ca oricare altă variabilă locală, atunci când se încheie funcţia.) Amintiţi-vă că o copie a obiectului există atât timp cât este executată funcţia. Aceasta înseamnă că acea copie poate să efectueze operaţii care, atunci când copia este distrusă, să necesite apelarea unei funcţii destructor. De exemplu, este perfect valid să se aloce memorie copiei, după care să se elibereze la distrugerea copiei. Din acest motiv, funcţia destructor trebuie executată când este distrusă copia.

În rezumat: funcţia constructor a unui obiect nu este apelată atunci când este creată o copie a obiectului prin transmiterea sa către o funcţie. Însă, atunci când este distrusă copia obiectului din interiorul funcţiei, este apelată funcţia ei destructor.

Implicit, când este făcută o copie a unui obiect, apare o copie la nivel de biţi. Aceasta înseamnă că noul obiect este un duplicat identic cu originalul. Acest fapt poate să devină, în timp, o sursă de neplăceri. Chiar dacă obiectele sunt pasate funcţiilor prin mecanismul normal de transmitere a parametrilor prin valoare care, teoretic, protejează şi izolează argumentul apelat, este încă posibilă apariţia efectelor secundare care pot afecta, sau chiar distruge, obiectul folosit ca argument. De exemplu, dacă unui obiect utilizat ca argument i se alocă memorie ce se eliberează când este distrus, atunci copia sa din interiorul funcţiei va elibera aceeaşi memorie când va fi apelat destructorul său. Aceasta va face ca obiectul iniţial să fie distrus efectiv, făcându-l inutilizabil. După cum veţi vedea mai departe în această carte, este posibil să preveniţi acest tip de probleme definind operaţiile de copiere în cadrul claselor dvs. prin crearea unui tip special de constructor numit *constructor de copie*. (Vedeţi **Capitolul 22.**)

Returnarea obiectelor

O funcţie poate returna un obiect rutinei care a apelat-o. De exemplu, acest program este valid în C++:

```
#include <iostream.h>

class clasamea {
    int i;
public:
    void pune_i(int n) {i=n;}
    int da_i() {return i;}
};

clasamea f(); // returneaza obiect de tipul clasamea

main()
```

```
{
    clasamea o;

    o = f();

    cout << o.da_i() << "\n";

    return 0;
}

clasamea f()
{
    clasamea x;
    x.pune_i(1);
    return x;
}
```

Când un obiect este returnat de o funcţie, este creat automat un obiect temporar, care conţine valoarea returnată. Acesta este, de fapt, obiectul care este returnat de către funcţie. După ce valoarea a fost returnată, acest obiect este distrus. Distrugerea obiectului temporar poate determina, în unele situaţii, efecte secundare neaşteptate. De exemplu, dacă obiectul care a fost returnat are un destructor care eliberează memoria dinamică alocată, acea memorie va fi eliberată chiar dacă obiectul care primeşte valoarea returnată încă o mai foloseşte. După cum veţi vedea mai departe, există căi de prevenire a acestei probleme care folosesc supraîncărcarea operatorului de atribuire şi definirea unui constructor de copii.

Atribuirea obiectelor

Presupunând că ambele obiecte sunt de acelaşi tip, puteţi să atribuiţi un obiect altuia. Aceasta va determina ca datele obiectului din membrul drept să fie copiate în datele obiectului din membrul stâng. De exemplu, următorul program va afişa **99**.

```
#include <iostream.h>

class clasamea {
    int i;
public:
    void pune_i(int n) {i=n;}
    int da_i() {return i;}
};
```



```
main()
{
    clasamea ob1, ob2;

    ob1.pune_i(99);
    ob2 = ob1; // atribuie datele din ob1 lui ob2

    cout << "acesta este i din ob2: " << ob2.da_i();

    return 0;
}
```

Implicit, toate datele dintr-un obiect sunt atribuite celuilalt, prin folosirea copierii bit-cu-bit. Dar, puteţi să supraîncărcaţi operatorul de atribuire şi să stabiliţi o altă procedură de atribuire (vedeţi **Capitolul 14**).

Capitolul 13

Matrice, pointeri şi referinţe

După cum ştiţi, pointerii şi rudele lor, matricele, sunt importanţi în limbajul C. De aceea, nu va fi o surpriză că aceştia rămân la fel de importanţi şi pentru facilităţile extinse asigurate de către C++. De fapt, pointerii sunt atât de importanţi în C++ încât a fost adăugată o nouă formă de pointeri, numită *referinţă*. Acest capitol studiază matricele, pointerii şi referinţele, relativ la obiecte.

Matrice de obiecte

Este posibil ca în C++ să aveţi matrice de obiecte. Sintaxa pentru declararea şi utilizarea unei astfel de matrice este exact aceeaşi ca şi pentru oricare alt tip de variabilă. De exemplu, acest program foloseşte o matrice de obiecte formată din trei elemente:

```
#include <iostream.h>

class cl {
    int i;
public:
    void pune_i(int j) {i=j;}
    int da_i() {return i;}
};

main()
{
    cl ob[3];
    int i;

    for(i=0; i<3; i++) ob[i].pune_i(i+1);

    for(i=0; i<3; i++)
        cout << ob[i].da_i() << "\n";

    return 0;
}
```

Programul afişează pe ecran numerele **1, 2** şi **3**.

Dacă o clasă defineşte un constructor cu parametri, puteţi iniţializa fiecare obiect din matrice indicând o listă de iniţializare în acelaşi mod în care o faceţi pentru alte tipuri de matrice. Însă forma exactă a listei de iniţializare depinde de numărul de parametri ceruţi de funcţia constructor a obiectului. Pentru obiectele al căror constructor are un singur parametru, puteţi pur şi simplu să specificaţi o listă de valori iniţiale folosind sintaxa normală de iniţializare a matricelor. Fiecare valoare din listă este transmisă în ordine funcţiei constructor pe măsură ce este creat fiecare element al matricei. De exemplu, iată o versiune uşor diferită a programului precedent, ce foloseşte iniţializarea:


```
#include <iostream.h>
class cl {
    int i;
public:
    cl(int j) {i=j;} // constructor
    int da_i() {return i;}
};
main()
{
    cl ob[3] = {1, 2, 3}; // initializare
    int i;
    for(i=0; i<3; i++)
        cout << ob[i].da_i() << "\n";
    return 0;
}
```

Şi acest program afişează pe ecran numerele **1, 2** şi **3**.

Dacă un constructor de obiecte necesită două sau mai multe argumente, atunci va trebui să utilizaţi o formă de iniţializare puţin diferită, prezentată aici:

```
#include <iostream.h>

class cl {
    int h;
    int i;
public:
    cl(int j, int k) {h=j; i=k; } // constructor
    int ia_i() {return i; }
    int ia_h() {return h; }
};

main()
{
    cl ob[3] = {
        cl(1, 2),
        cl(3, 4),
        cl(5, 6)
    };    // initializare

    int i;
```

```
    for(i=0; i<3; i++) {
        cout << ob[i].ia_h();
        cout << ",";
        cout << ob[i].ia_i() << "\n";
    }
    return 0;
}
```

Constructorul lui **cl** are, în acest exemplu, doi parametri şi, de aceea, necesită două argumente. Aceasta înseamnă că forma „prescurtată" nu poate fi folosită. În locul ei, utilizaţi forma „lungă" prezentată în exemplu. Desigur, puteţi folosi forma lungă şi în cazurile în care constructorul necesită doar un singur argument. Numai că forma scurtă este mai uşor de utilizat.

Matrice iniţializate / matrice neiniţializate

O situaţie de clasă specială se întâlneşte atunci când doriţi să creaţi atât matrice iniţializate cât şi neiniţializate. Să considerăm următoarea **class**:

```
class cl {
    int i;
public:
    cl(int j) {i=j;}
    int da_i() {return i;}
};
```

Funcţia constructor definită aici de **cl** necesită un parametru. Aceasta înseamnă că orice matrice declarată ca fiind de acest tip trebuie iniţializată, deci următoarea declaraţie de matrice este greşită:

```
cl a[9]; // eroare, constructorul necesita initializare
```

Motivul pentru care instrucţiunea nu este corectă (aşa cum este definită **cl**) constă în aceea că această clasă ar avea un constructor fără parametri, deoarece nu este specificată nici o iniţializare. Totuşi, aşa cum s-a afirmat, **cl** nu are constructor de parametri. Deoarece nu există nici un constructor valabil care să corespundă acestei declaraţii, compilatorul va afişa o eroare.

Pentru a rezolva această problemă, este necesar să supraîncărcaţi funcţia constructor, adăugând încă una care nu preia nici un parametru. În acest fel, sunt admise atât matricele care sunt iniţializate, cât şi acelea care nu sunt iniţializate. (Supraîncărcarea este discutată în detaliu în **Capitolul 14**.) Iată, de exemplu, o versiune îmbunătăţită a lui **cl**:


```
class cl {
    int i;
public:
    cl() {i=0;} // apelare pentru matrice neinitializate
    cl(int j) {i=j;} // apelare pentru matrice initializate
    int da_i() {return i;}
};
```

Fiind date aceste **class**, sunt permise ambele instrucţiuni care urmează:

```
cl a1[3] = {3, 5, 6};      // initializat
cl a2[34];       // neinitalizat
```

Pointeri către obiecte

Exact aşa cum puteţi avea pointeri către alte tipuri de variabile, puteţi avea pointeri către obiecte. Când doriţi acces la membrii unei clase cu ajutorul unui pointer către un obiect, folosiţi operatorul săgeată (->) în locul operatorului punct. Următorul program ilustrează cum se poate căpăta acces la un obiect, dacă există un pointer către el:

```
#include <iostream.h>

class cl {
    int i;
public:
    cl(int j) {i=j;}
    int da_i() {return i;}
};

main()
{
    cl ob(88), *p;

    p = &ob; // da adresa lui ob

    cout << p->da_i(); // foloseste -> pentru a apela da_i()

    return 0;
}
```

După cum ştiţi, atunci când este incrementat un pointer, el indică spre următorul element de acelaşi tip cu al său. De exemplu, un pointer de tip întreg va indica spre următorul întreg. În general, întreaga aritmetică a pointerilor este relativă la tipul de bază al acestora, la tipul de date către care indică pointerul. Acelaşi lucru este valabil pentru pointeri spre obiecte. De exemplu, următorul program foloseşte un pointer pentru a obţine acces la toate trei elementele matricei **ob** după ce i-a fost atribuită adresa de început din **ob**.

```
#include <iostream.h

class cl {
    int i;
public:
    cl() {i=0;}
    cl(int j) {i=j;}
    int da_i() {return i;}
};

main()
{
    cl ob [3] = {1, 2, 3};
    cl *p;
    int i;

    p = ob; // preia inceputul matricei
    for(i=0; i<3; i++) {
          cout << p->da_i() << "\n";
          p++; // indica spre urmatorul obiect
    }

    return 0;
}
```

Puteţi să atribuiţi unui pointer adresa unui membru public al unui obiect şi apoi să aveţi acces la acel membru folosind pointerul. De exemplu, acesta este un program în C++ care afişează pe ecran numărul **1**:

```
#include <iostream.h>

class cl {
public:
    int i;
    cl(int j) {i=j;}
```



```
};

main()
{
    cl ob(1);
    int *p;

    p = &ob.i; // preia adresa lui ob.i

    cout << *p; // are acces la ob.i prin p

    return 0;
}
```

Deoarece **p** indică spre un întreg, el este declarat ca un pointer de tip întreg. În această situaţie este irelevant că **i** este un membru al obiectului **ob**.

Pointeri de verificare a tipului în C++

Trebuie să înţelegeţi un lucru important despre pointeri în C++: puteţi să atribuiţi un pointer altuia doar dacă cei doi au tipuri compatibile. De exemplu, dacă se dă

```
int *pi;
float *pf;
```

următoarea atribuire este ilegală în C++:

```
pi = pf; // eroare -- nepotrivire de tipuri
```

Desigur, puteţi să eliminaţi orice incompatibilitate folosind un modelator, dar procedând astfel violaţi mecanismul de verificare a tipului din C++.

NOTĂ: Verificarea mai strictă a tipurilor în C++, atunci când sunt implicaţi pointeri, diferă de C, în care puteţi atribui orice tip de valoare oricărui pointer.

Pointerul *this*

Când este apelată o funcţie membru, i se pasează automat un argument implicit, care este un pointer către obiectul care a generat apelarea (obiectul care a invocat funcţia). Acest pointer este numit **this**. Pentru a-l înţelege pe **this**, să luăm un

program care creează o clasă numită **putere**, ce calculează rezultatul dat de ridicarea unui număr la o putere:

```
#include <iostream.h>

class putere {
    double b;
    int e;
    double val;
public:
    putere(double baza, int exp);
    double da_put() {return val;}
};

putere::putere(double baza, int exp)
{
    b = baza;
    e = exp;
    val = 1;
    if(exp==0) return;
    for( ; exp>0; exp--) val = val * b;
}

main()
{
    putere x(4.0, 2), y(2.5, 1), z(5.7, 0);

    cout << x.da_put() << " ";
    cout << y.da_put() << " ";
    cout << z.da_put() << "\n";

    return 0;
}
```

La membrii unei **class** se poate căpăta acces direct din cadrul unei funcţii membru, fără nici un specificator de obiect sau de clasă. De aceea, instrucţiunea

```
b = baza;
```

din interiorul funcţiei **putere()**, comandă ca valoarea conţinută în **baza** să fie atribuită unei copii a lui **b** asociată obiectului care a generat apelarea. Totuşi, aceeaşi instrucţiune poate fi scrisă şi astfel:

```
this->b = baza;
```


Amintiţi-vă că pointerul **this** indică spre obiectul care a apelat **putere()**. Astfel, **this->b** se referă la copia b pentru acel obiect. De exemplu, dacă **putere()** este apelată de **x** (ca în **x(4.0, 2)**), atunci **this** din instrucţiunea precedentă indică spre **x**. Reţineţi că scrierea instrucţiunii fără **this** este doar o prescurtare.

Iată întreaga funcţie **putere()** scrisă folosind pointerul **this**:

```
putere::putere(double baza, int exp)
{
    this->b = baza;
    this->e = exp;
    this->val = 1;
    if(exp==0) = 1;
    for( ; exp>0; exp--)
        this->val = this->val * this->b;
```

Nici un programator de C++ nu va scrie, de fapt, **putere()** aşa cum tocmai am arătat deoarece nu se câştigă numic, iar forma prescurtată este mai uşoară. Totuşi, pointerul **this** este foarte important la supraîncărcarea operatorilor (vedeţi Capitolul 14) şi ori de câte ori o funcţie membru trebuie să utilizeze un pointer către obiectul care a apelat-o.

Amintiţi-vă că pointerul **this** este transmis automat către toate funcţiile membru. De aceea, **da_put()** poate fi rescrisă astfel:

```
double da_put() {return this->val;}
```

În acest caz, dacă **da_put()** este apelată astfel:

```
y.da_put();
```

atunci **this** va indica spre obiectul y.

Încă două lucruri despre **this**. Primul, funcţiile **friend** nu sunt membri ai clasei şi, de aceea, nu le sunt pasaţi pointeri **this**. Al doilea, funcţiile membre **static** nu au un pointer **this**.

Pointeri către tipuri derivate

În general, un pointer de un anumit tip nu poate indica spre un obiect de un tip diferit. Totuşi, există o excepţie importantă la această regulă care se referă doar la clasele derivate. Pentru început, să presupunem că avem două clase numite **B** şi **D**. Să mai presupunem că **D** este derivat din clasa de bază **B**. În această situaţie, un pointer de tip **B*** poate să indice şi spre un obiect de tip **D**. Mai general, un pointer din clasa de bază poate să fie folosit ca un pointer spre un obiect din oricare clasă derivată din acea bază.

Chiar dacă un pointer din clasa de bază poate indica spre un obiect derivat, reciproca nu este adevărată. Un pointer de tip D* nu poate indica spre un obiect de tip B. Mai mult, chiar dacă puteţi să folosiţi un pointer din bază pentru a indica un obiect derivat, puteţi avea acces doar la membrii de tipul derivat care au fost importaţi din bază. Aceasta înseamnă că nu veţi fi putea avea acces la nici unul din membrii adăugaţi de clasa derivată. (Puteţi însă să convertiţi un pointer din bază într-unul derivat şi să câştigaţi acces deplin la întreaga clasă derivată.)

Iată un scurt program care ilustrează această facilitate din C++:

```
#include <iostream.h>

class baza {
    int i;
public:
    void pune_i(int num) {i=num;}
    int da_i() {return i;}
};

class derivat: public baza {
    int j;
public:
    void pune_j(int num) {j=num;}
    int da_j() {return j;}
};

main()
{
    baza *bp;
    derivat d;

    bp = &d; // pointerul din baza indica spre obiectul derivat

    // acces la obiectul derivat folosind pointerul din baza
    bp->pune_i(10);
    cout << bp->da_i() << " ";

/* Urmatoarele comenzi nu vor functiona. Nu puteti avea
   acces la un element al clasei derivate folosind un
   pointer din clasa de baza.
    bp->pune_j(88);  // eroare
    cout << bp->da_j(); // eroare

 */
```



```
    return 0;
}
```

După cum puteţi vedea, un pointer din bază este folosit pentru a oferi acces la un obiect dintr-o clasă derivată.

Pentru a avea acces la un membru al clasei derivate prin intermediul unui pointer de bază, puteţi să-i convertiţi într-altul, al tipului derivat, chiar dacă acest lucru este considerat ca fiind neprofesionist de către majoritatea programatorilor în C++. De exemplu, acesta este un cod valid în C++:

```
// accesul este acum permis datorita conversiei
((derivat *)bp)->pune_j(88);
cout << ((derivat *)bp)->da_j();
```

Este important să vă reamintiţi că aritmetica pointerilor este relativă la tipul de bază al pointerului. Din acest motiv, pentru un pointer din bază, ce indică spre un obiect derivat, incrementarea nu îl determină să indice spre următorul obiect de tipul derivat. În loc de aceasta, el va indica spre ceea ce crede că este următorul obiect de tipul de bază. De exemplu, următorul program, deşi este corect sintactic, conţine această eroare:

```
#include <iostream.h>

class baza {
    int i;
public:
    void pune_i(int num) {i=num;}
    int da_i() {return i;}
};

class derivat: public baza {
    int j;
public:
    void pune_j(int num) {j=num;}
    int da_j() {return j;}
};

main()
{
    baza *bp;
    derivat d[2];
```

```
    bp = d;

    d[0].pune_i(1);
    d[1].pune_i(2);

    cout << bp->da_i() << " ";
    bp++; // relativ la baza, nu la clasa derivata
    cout << bp->da_i(); // afiseaza o valoare gresita

    return 0;
}
```

Utilizarea pointerilor din bază către tipuri derivate este foarte folositoare când creați polimorfism în timpul rulării prin mecanismul funcțiilor virtuale (vedeți Capitolul 16).

Pointeri către membrii clasei

C++ permite să generați un tip special de pointeri care „indică" generic către un membru al unei clase, nu către un anumit exemplar al acelui membru dintr-un obiect. Acest tip de pointer este numit un pointer către un membru al clasei sau, pe scurt, un *pointer-la-membru*. În C++ un pointer la membru nu este același lucru cu un pointer normal. Un pointer la un membru asigură doar un offset (o poziție) într-un obiect din clasa membrului, unde poate fi găsit acel membru. Deoarece pointerii la membri nu sunt pointeri adevărați, nu li se pot aplica operatorii **.** și **->**. Pentru a avea acces la membrul unei clase prin intermediul unui pointer spre el, va trebui să folosiți operatorii speciali ai pointerilor la membri, **.*** și **->***. Misiunea lor este să vă permită accesul la un membru al unei clase prin intermediul unui pointer către acesta.

Iată un exemplu:

```
#include <iostream.h>

class cl{
public:
    cl(int i) {val=i;}
    int val;
    int val_dubla() {return val+val;}
};

main()
{
    int cl::*date;    // pointer la o data membru
```



```
    int (cl::*func)(); // pointer la o functie membru
    cl ob1(1), ob2(2); // creeaza obiecte

    date = &cl::val; // da offsetul pentru val
    func = &cl::val_dubla; // da offsetul pentru val_dubla()

    cout << "Iata valorile: ";
    cout << ob1.*date << " " << ob2.*date << "\n";
    cout << "Aici ele sunt dublate: ";
    cout << (ob1.*func)() << " ";
    cout << (ob2.*func)() << "\n";

    return 0;
}
```

În **main()** acest program creează doi pointeri la membri: **date** și **func**. Notați cu atenție sintaxa fiecărei declarații. Când declarați pointeri la membri, trebuie să specificați clasa și să folosiți operatorul de specificare a domeniului. Programul creează, de asemenea, obiectele de tip **cl** numite **ob1** și **ob2**. După cum ilustrează programul, pointerii la membri pot să indice atât funcții cât și date. Apoi programul obține adresele lui **val** și **val_dubla()**. După cum am spus mai devreme, aceste „adrese" sunt, de fapt, offseturi într-un obiect de tip **cl**, la care pot fi găsite **val** și **val_dubla()**. În continuare, pentru a afișa valorile oricăror obiecte de tip **val**, fiecare valoare este accesată prin **date**. În sfârșit, programul folosește **func** pentru a apela funcția **val_dubla()**. Parantezele în plus sunt necesare pentru corecta asociere a operatorului **.***.

Când cereți acces la un membru al unui obiect folosind un obiect sau o referință (discutată mai târziu în acest capitol), trebuie să folosiți operatorul **.***. Dar, dacă utilizați un pointer spre un obiect, va trebui să folosiți operatorul **->***, așa cum se ilustrează în următoarea versiune a programului precedent.

```
#include <iostream.h>

class cl{
public:
    cl(int i) {val=i;}
    int val;
    int val_dubla() {return val+val;}
};

main()
{
    int cl::*date;    // pointer la o data membru
```

```
    int (cl::*func)(); // pointer la o functie membru
    cl ob1(1), ob2(2); // creeaza obiecte
    cl *p1, *p2;

    p1 = &ob1;
    p2 = &ob2;

    date = &cl::val; // da offsetul pentru val
    func = &cl::val_dubla; // da offsetul pentru val_dubla()

    cout << "Iata valorile: ";
    cout << p1->*date << "" << p2->*date << "\n";

    cout << "Aici ele sunt dublate: ";
    cout << (p1->*func)() << " ";
    cout << (p2->*func)() << "\n";

    return 0;
}
```

În această versiune, **p1** și **p2** sunt pointeri la obiecte de tip **cl**. De aceea, operatorul **->*** este folosit pentru a oferi acces la **val** și la **val_dubla()**.

Amintiți-vă că pointerii la membri sunt diferiți de pointerii spre exemplare efective de elemente ale unui obiect. De exemplu, să considerăm următorul fragment. (Să presupunem că s-a declarat **cl** după cum s-a arătat în programul precedent.)

```
int cl::*d;
int *p;
cl o;

p = &o.val // aceasta este adresa unui val efectiv
d = &cl::val // acesta este offsetul lui val generic
```

Aici, p este un pointer la un întreg din interiorul unui obiect *efectiv*, pe când **d** este un simplu offset care indică unde se găsește **val** în orice obiect de tip **cl**.

Operatorii pointer-la-membru se aplică în situații speciale. Ei nu sunt folosiți curent în programarea de zi cu zi.

Referințe

C++ conține o caracteristică ce este legată de pointeri. Aceasta este numită referință. O *referință* este, în esență, un pointer implicit, care acționează ca un alt nume al unui obiect.

Parametri de referință

O utilizare importantă a unei referințe este să vă permită să creați funcții care folosesc automat transmiterea parametrului prin referință și nu metoda implicită în C++, cea de apelare prin valoare.

După cum știți, pentru a crea o apelare prin referință în C, trebuie să pasați explicit funcției adresa argumentului. De exemplu, să considerăm următorul scurt program, care folosește această abordare într-o funcție numită **neg()**, ce inversează semnul variabilei de tip întreg spre care indică argumentul ei.

```
#include <iostream.h>

void neg(int *i);

main()
{
    int x;

    x = 10;
    cout << x << " este negat ";

    neg(&x);
    cout << x << "\n";

    return 0;
}
void neg(int *i)
{
    *i = -*i;
}
```

În acest program, **neg()** preia ca parametru un pointer către un întreg al cărui semn îl va schimba. De aceea, **neg()** este apelat explicit cu adresa lui **x**. Mai departe, în interiorul lui **neg()**, pentru a avea acces la variabila spre care indică **i**, trebuie folosit operatorul *. După cum știți, acesta este modul de generare a unei apelări-prin-referire manuale. Dar, în C++, puteți automatiza această operație folosind un parametru de referință.

Pentru a crea un parametru de referință, precedați numele parametrului cu un **&**. Iată cum este declarat **neg()** prin referință:

```
void neg(int &i);
```

Asta spune compilatorului să îl transforme pe **i** ca parametru de referință. O dată făcută aceasta, **i** devine, practic, un alt nume pentru orice argument folosit la

apelul funcţiei **neg()**. Cu alte cuvinte, i este un pointer implicit care se referă automat la argumentul folosit pentru invocarea funcţiei **neg()**. O dată transformat i ca referinţă, nu mai este necesar (şi nici chiar permis) să aplicaţi operatorul *. De fiecare dată când este folosit, i este implicit o referinţă la argument. Mai mult, la apelul funcţiei **neg()**, nu mai este necesar (şi nici chiar permis) să precedaţi numele argumentului cu operatorul **&**. Compilatorul va face aceasta automat. Iată o versiune cu referinţă a precedentului program:

```
#include <iostream.h>

void neg(int &i); // i este acum o referinta

main()
{
    int x;

    x = 10;
    cout << x << "este negat"

    neg(x); // nu mai este nevoie de operatorul &
    cout << x <<"\n";

    return 0;
}

void neg(int &i)
{
    i = -i; // i este acum o referinta, nu are nevoie de *
}
```

Să recapitulăm: când creaţi un parametru de referinţă, acesta se referă automat la (indică implicit spre) argumentul folosit pentru a apela funcţia. De aceea, instrucţiunea

```
i = -i;
```

operează efectiv asupra lui **x**, nu asupra copiei lui **x**. Nu este necesar să aplicaţi unui argument operatorul **&**. De asemenea, în interiorul funcţiei, parametrul este folosit direct, fără să aibă nevoie de operatorul *.

Este important să înţelegeţi că atunci când atribuiţi o valoare referinţei, de fapt atribuiţi efectiv acea valoare variabilei căreia i se adresează referinţa. În cazul parametrilor funcţiei, aceasta va fi variabila folosită în apelarea funcţiei.

Nu este posibil să modificaţi în interiorul funcţiei ceea ce „indică" parametrul. Aceasta înseamnă că o instrucţiune ca:

```
i++;
```

aflată în interiorul funcţiei **neg()**, incrementează valoarea variabilei folosită pentru apelare. Ea nu determină ca i să indice spre o nouă locaţie.

Iată un alt exemplu. Acest program foloseşte parametri de referinţă pentru a inversa între ele valorile variabilelor cu care este apelată funcţia. (Funcţia **schimb()** este exemplul clasic de transmitere a parametrilor prin referinţă.)

```
#include <iostream.h>

void schimb(int &i, int &j);

main()
{
    int a, b, c, d;
    a = 1;
    b = 2;
    c = 3;
    d = 4;

    cout << "a si b: " << a << " " << b << "\n";
    schimb(a, b); // nu este necesar operatorul &
    cout << " a si b: " << a << " " << b << "\n";

    cout << "c si d: " << c << " " << d << "\n";
    schimb(c, d);
    cout << "c si d: " << c << " " << d << "\n";

    return 0;
}
void schimb(int &i, int &j)
{
    int t;

    t = i; // nu este necesar operatorul *
    i = j;
    j = t;
}
```

Acest program afişează următoarele:

```
a si b: 1 2
a si b: 2 1
c si d: 3 4
c si d: 4 3
```

Transmiterea referinţelor către obiecte

În **Capitolul 12** a fost explicat că atunci când un obiect este transmis unei funcţii ca argument, se face o copie a acelui obiect. Mai mult, când se face copia, *nu* este apelată funcţia constructor obişnuită. (În locul acesteia se face o copie exactă a argumentului apelant.) În schimb, când funcţia se încheie, este apelat destructorul copiei. Dacă, din anumite motive, nu doriţi să fie apelată funcţia destructor, transmiteţi obiectul prin referinţă. (Mai târziu, în acest capitol, veţi vedea asemenea exemple.) Când apelaţi prin referinţă, nu se face nici o copie a obiectului. Aceasta înseamnă că nici un obiect folosit ca parametru nu este distrus atunci când se termină funcţia, iar destructorul parametrului nu este apelat. Încercaţi, de exemplu, acest program:

```
#include <iostream.h>

class cl {
    int ex;
public:
    int i;
    cl(int i);
    ~cl();
    void neg(cl &o) {o.i = -o.i;} // nu este creat temporar
};

cl::cl(int num)
{
    cout << "Construieste " << num << "\n";
    ex = num;
}

cl::~cl()
{
    cout << "Distruge " << ex << "\n";
}

main()
{
    cl o(1);

    o.i = 10;
    o.neg(o);

    cout << o.i << "\n";

    return 0;
}
```

Iată ieşirea programului:

```
Construieste 1
-10
Distruge 1
```

După cum puteţi vedea, funcţia destructor a lui cl este apelată o singură dată. Dacă o ar fi apelat prin valoare, atunci în interiorul lui **neg()** s-ar fi creat al doilea obiect, iar destructorul ar fi fost apelat a doua oară când obiectul ar fi fost distrus la terminarea lui **neg()**.

Când parametrii sunt transmişi prin referinţă, amintiţi-vă că schimbările obiectului din interiorul funcţiei afectează obiectul apelant.

Returnarea referinţelor

O funcţie poate să returneze o referinţă. Aceasta are efectul neaşteptat de a permite unei funcţii să fie folosită în membrul stâng al unei instrucţiuni de atribuire! De exemplu, să luăm acest program simplu:

```
#include <iostream.h>

char &inloc(int i); // returneaza o referinta

char s[80] = "Va salut";

main()
{
    inloc(2) = 'X'; // atribuie X spatiului de dupa Va
    cout << s;
    return 0;
}

char &inloc(int i)
```

```
    {
        return s[i];
    }
```

Acest program înlocuieşte spaţiul dintre **Va şi salut** cu un **X**. Aceasta înseamnă că programul afişează **VaXsalut**. Să vedem cum se realizează aceasta.

După cum am arătat, **inloc()** este declarată ca returnând o referinţă către o matrice de tip caracter. După cum este scris **inloc()**, ea returnează o referinţă către elementul şirului **s**, specificat de argumentul său **i**. Referinţa returnată de **inloc()** este folosită apoi în **main()** pentru a atribui acelui element caracterul **X**.

Referinţe independente

De departe cea mai obişnuită utilizare pentru referinţe este pasarea argumentelor prin referinţă şi obţinerea valorii returnate de funcţie. Totuşi, puteţi declara o referinţă care să fie doar o simplă variabilă. Acest tip de referinţă este numită *referinţă independentă*.

Când creaţi o referinţă independentă, tot ceea ce creaţi este un al doilea nume pentru altă variabilă. Toate variabilele de tip referinţă independentă trebuie iniţializate atunci când le creaţi. Motivul este uşor de înţeles. În afara iniţializării nu puteţi modifica obiectul spre care indică variabila de referinţă. De aceea, ea trebuie iniţializată atunci când este declarată. (În C++, iniţializarea este o operaţie complet separată de atribuire.)

Programul următor ilustrează o referinţă independentă.

```
#include <iostream.h>

main()
{
    int a;
    int &ref = a; // referinta independenta

    a = 10;
    cout << a << " " << ref << "\n";

    ref = 100;
    cout << a << " " << "\n";

    int b = 19;
    ref = b; // aceasta pune valoarea b in a
    cout << a << " " << ref << "\n";

    ref--; // acesta decrementeaza a
        // nu afecteaza la ce se refera ref
    cout << a << " " << ref << "\n";

    return 0;
}
```


Programul afişează această ieşire:

```
10 10
100 100
19 19
18 18
```

Pentru a vă referi la o constantă, puteţi folosi o referinţă independentă. De exemplu:

```
int &count = 9;
```

determină **count** să indice localizarea valorii 9 în tabelul de constante din programul dvs.

De fapt, referinţele independente sunt de mică valoare practică deoarece fiecare din ele este, ad literam, doar un alt nume pentru altă variabilă. Având două nume care descriu acelaşi obiect, probabil că programul dvs. va fi confuz şi dezorganizat.

Restricţii pentru referinţe

Există mai multe restricţii care se aplică referinţelor. Nu vă puteţi referi la o altă referinţă. Altfel spus, nu puteţi obţine adresa unei referinţe. Nu puteţi să creaţi referinţă. Nu puteţi să creaţi un pointer spre o referinţă. Nu puteţi să vă matrice de referinţe. Nu puteţi să creaţi un pointer spre o referinţă. Nu puteţi să vă referiţi la un câmp de biţi.

O variabilă de tip referinţă trebuie să fie iniţializată când este declarată, doar dacă nu este un membru al unei clase, un parametru de funcţie sau o valoare returnată. Referinţele nule sunt interzise.

O problemă de stil

Când declară variabile tip pointer sau referinţă, unii programatori de C++ folosesc un stil propriu care asociază * sau & cu numele tipului şi nu cu al variabilei. Iată, de exemplu, două declaraţii echivalente funcţional:

```
int& p; // & asociat cu tipul
int &p; // & asociat cu variabila
```

Asocierea lui * sau & cu numele tipului reflectă dorința unor programatori ca C++ să conțină separat un pointer pentru tip. Dar necazul asocierii operatorilor & și * cu numele tipului, și nu cu cel al variabilei, este că, în conformitate cu sintaxa formală din C++, nici & și nici * nu sunt distributivi într-o listă de parametri. De aceea, se pot crea ușor declarații greșite. Următoarea declarație creează, de exemplu, *unul, nu doi* pointeri de tip întreg. Aici, **b** este declarat ca un întreg (nu ca un pointer de tip întreg) deoarece, conform sintaxei din C++, când este folosit într-o declarație, * (sau &) este atașat variabilei individuale pe care o precede, nu tipului căruia îi urmează.

```
int* a, b;
```

Problema acestei declarații este că, văzând mesajul, ni se sugerează că atât **a** cât și **b** sunt de tip pointer, chiar dacă, de fapt, doar **a** este un pointer. Această confuzie vizuală nu păcălește doar programatorii novici în C++, ci uneori și pe cei cu experiență.

Este important să înțelegeți că, în ceea ce privește compilatorul de C++, nu are importanță dacă scrieți **int *p** sau **int* p**. De aceea, dacă doriți să asociați * sau & cu tipul și nu cu variabila, sunteți liberi să o faceți. Totuși, pentru a evita confuzia, această carte va continua să asocieze * și & cu variabilele pe care le modifică și nu cu tipurile lor.

Operatorii de alocare dinamică din C++

În C, alocarea dinamică a memoriei este realizată prin funcțiile **malloc()** și **free()**. De dragul compatibilității, funcțiile de alocare dinamică din C sunt valabile și în C++. Dar, C++ asigură un sistem propriu alternativ de alocare dinamică, bazat pe doi operatori: **new** și **delete**. După cum veți vedea, există avantaje substanțiale ale facilităților de C++ pentru alocarea dinamică a memoriei.

Operatorul **new** returnează un pointer către memoria alocată. Ca și **malloc()**, **new** alocă memorie din heap (zona de memorie liberă). Dacă nu există memorie suficientă pentru a acoperi alocarea cerută, se returnează un pointer null. Operatorul **delete** eliberează memoria alocată anterior prin utilizarea operatorului **new**. Formele generale pentru **new** și **delete** sunt:

p_var = new *tip;*
delete *p_var*;

Aici, *p_var* este o variabilă de tip pointer care primește un pointer spre memoria care este suficient de mare pentru a păstra un element de tipul *tip*. Iată, de exemplu, un program care alocă memorie pentru a înregistra un întreg:


```
#include <iostream.h>
#include <stdlib.h>

main()
{
    int *p;

    p = new int; // aloca spatiu pentru un int

    if(!p) {
        cout << "Eroare de alocare\n";
        exit(1);
    }

    *p = 100;

    cout << "La " << p << " ";
    cout << " este valoarea " << *p << "\n";

    delete p;
    return 0;
}
```

Operatorul **delete** trebuie să fie folosit doar cu un pointer valid, alocat deja prin utilizarea lui **new**. Folosirea oricărui alt tip de pointer cu **delete** este o operațiune cu rezultat nedefinit și, aproape sigur, va determina probleme serioase, cum ar fi căderea sistemului.

Deși **new** și **delete** efectuează funcții similare cu **malloc()** și **free()**, ele au mai multe avantaje. Mai întâi, **new** alocă automat memorie suficientă pentru a păstra obiectele de tipul specificat. Nu este necesar să folosiți operatorul **sizeof**. Deoarece mărimea este calculată automat, se elimină orice posibilitate de eroare în această privință. Apoi, **new** returnează automat un pointer de tipul specificat. Nu este necesar să folosiți explicit un modelator de tip, așa cum o făceați când alocați memorie cu **malloc()**. În sfârșit, atât **new** cât și **delete** pot fi supraîncărcate, permițându-vă să creați un sistem de alocare propriu.

Puteți să inițializați memoria alocată cu o valoare cunoscută, scriind în instrucțiunea **new** o valoare după numele tipului. Iată forma generală a operatorului **new** când include și inițializare:

p_var = new *tip_var* (*inițializator*)

De exemplu, în următorul program întregul căruia i se alocă memorie capătă inițial valoarea 87.

```
#include <iostream.h>
#include <stdlib.h>

main()
{
    int *p;

    p = new int (87); // initializeaza cu 87

    if(! p) {
          cout << "Eroare de alocare\n";
          exit(1);
    }

    cout << "La " << p << " ";
    cout << "este valoarea " << *p << "\n";

    delete p;

    return 0;
}
```

Folosind **new** cu următoarea formă generală puteţi aloca memorie matricelor:

p_var = new *tip_matrice* [*marime*];

Pentru a elibera memoria pentru o matrice, folosiţi această formă pentru **delete**:

delete [ ] *p_var*;

Aici, [ ] spune operatorului **delete** că va fi eliberată o matrice.
De exemplu, următorul program atribuie memorie pentru o matrice cu zece elemente:

```
#include <iostream.h>
#include <stdlib.h>

main()
{
    int *p, i;

    p = new int [10]; /* aloca memorie pentru o matrice de
                         10 elemente */
```



```
    if(!p) {
          cout << "eroare de alocare\n";
          exit(1);
    }

    for(i=0; i<10; i++)
          p[i];

    for(i=0; i<10; i++;
          cout << p[i] << " ";

    delete [ ] p; // elibereaza memoria pentru matrice

    return 0;
}
```

Observaţi instrucţiunea **delete**. Cum s-a menţionat, când este eliberată memoria alocată cu **new** unei matrice, **delete** trebuie să fie pus în gardă că va fi eliberată o matrice, folosindu-se [ ]. (Cum veţi vedea în următorul paragraf, lucrul prezintă importanţă, în special când alocaţi memorie pentru matrice de obiecte.)
Alocării matricelor i se aplică o restricţie: ele nu pot primi valori iniţiale. Aceasta înseamnă că atunci când alocaţi memorie matricelor, nu puteţi specifica o iniţializare.

Alocarea de memorie obiectelor

Obiectelor li se poate aloca memorie dinamic, folosindu-se **new**. Când faceţi aceasta, se creează un obiect şi se returnează un pointer către el. Obiectul creat dinamic se comportă ca şi oricare alt obiect. Când este creat, este apelată funcţia sa constructor (dacă are una). Când este eliberat, se execută funcţia destructor.
Iată un program scurt care creează o clasă numită **bilant** şi care asociază numele unei persoane cu bilanţul său. În interiorul funcţiei **main()** este creat dinamic un obiect de tip **bilant()**.

```
#include <iostream.h>
#include <stdlib.h>
#include <string.h>

class bilant {
    double bil_crt;
    char nume[80];
public:
```

```
    void pune(double n, char *s) {
          bil_crt = n;
          strcpy(nume, s);
    }
    void ia_bil(double &n, char *s) {
          n = bil_crt;
          strcpy(s, nume);
    }
};

main()
{
    bilant *p;
    char s[80];
    double n;

    p = new bilant;
    if(!p) {
          cout << "Eroare de alocare\n";
          exit(1);
    }

    p->pune(12387.87, "Ralph Wilson");

    p->ia_bil(n, s);

    cout << s << " are bilantul: " << n;
    cout << "\n";

    delete p;

    return 0;
}
```

Deoarece **p** conţine un pointer către un obiect, operatorul săgeată este folosit pentru a oferi acces la membrii obiectului.

După cum am spus, obiectele alocate dinamic pot avea constructori şi destructori. Funcţiile constructor pot fi parametrizate. Să examinăm următoarea versiune a programului anterior:

```
#include <iostream.h>
#include <stdlib.h>
#include <string.h>
```



```
class bilant {
    double bil_crt;
    char nume[80];
public:
    bilant(double n, char *s) {
          bil_crt = n;
          strcpy(nume, s);
    }
    ~bilant() {
          cout << "Destructor ";
          cout << nume << "\n";
    }
    void ia_bil(double &n, char *s) {
          n = bil_crt;
          strcpy(s, nume);
    }
};

main()
{
    bilant *p;
    char s[80];
    double n;

    // Aceasta versiune foloseste o initializare
    p = new bilant (12387.87, "Ralph Wilson");
    if(!p) {
          cout << "Eroare de alocare\n";
          exit(1);
    }

    p->ia_bil(n, s);

    cout << s << " are bilantul: " << n;
    cout << "\n";

    delete p;

    return 0;
}
```

Parametrii funcţiei constructor ai obiectului sunt specificaţi după numele tipului, la fel ca şi pentru alte tipuri de iniţializări.

Puteţi să alocaţi memorie pentru matricele de obiecte, dar există o şmecherie. Deoarece nici o matrice căreia i se aplică **new** nu poate fi iniţializată, trebuie să fiţi siguri că, în cazul în care clasa conţine funcţii constructor, una va fi fără parametri. Dacă nu o faceţi, compilatorul de C++ nu va găsi un constructor corespunzător când va încerca să aloce memorie matricei şi nu va compila programul dvs.

În această versiune a programului precedent se alocă memorie unei matrice de obiecte tip **bilant** şi este apelat constructorul fără parametri.

```
#include <iostream.h>
#include <stdlib.h>
#include <string.h>

class bilant {
    double bil_crt;
    char nume[80];
public:
    bilant(double n, char *s) {
          bil_crt = n;
          strcpy(nume, s);
    }
    bilant() {} // constructor fara parametri
    ~bilant() {
          cout << "Destructor ";
          cout << nume << "\n";
    }
    void pune(double &n, char *s) {
          bil_crt = n;
          strcpy(nume, s);
    }
    void ia_bil(double &n, char *s) {
          n = bil_crt;
          strcpy(s, nume);
    }
};

main()
{
    bilant *p;
    char s[80];
    double n;
    int i;

    p = new bilant [3]; // aloca memorie pentru intreaga matrice

    if(!p) {
          cout << "Eroare de alocare\n";
          exit(1);
    }

    // retineti folosirea punctului, în loc de operatorul sageata
    p[0].pune(12387.87, "Ralph Wilson");
    p[1]. pune(144.00, "A.C. Conners");
    p[2]. pune(-11.23, "I.S. Falitu");

    for(i=0; i<3; i++) {
          p[i].ia_bil(n, s);

    cout << s << " are bilantul: " << n;
    cout << "\n";
    }

    delete [ ] p;

    return 0;
}
```

Un motiv pentru care aveţi nevoie să folosiţi forma **delete [ ]** când distrugeţi o matrice de obiecte alocate dinamic este acela că funcţia destructor poate fi apelată pentru fiecare obiect al matricei.

O notă finală referitoare la new şi **delete**: chiar dacă nu există o regulă care să stabilească acest lucru, este mai bine să nu combinaţi **new** şi **delete** cu **malloc()** şi **free()** în acelaşi program. Nu există nici o garanţie că ele sunt mutual compatibile.

Capitolul 14

Supraîncărcarea funcţiilor şi a operatorilor

C++

Supraîncărcarea funcţiilor şi a operatorilor (overloading) sunt esenţiale pentru programarea în C++. Nu numai că aceste două caracteristici oferă o bază importantă pentru polimorfismul din timpul compilării, dar, de asemenea, ele adaugă limbajului flexibilitate şi extensibilitate. De exemplu, supraîncărcarea operatorilor << şi >> constituie baza abordării I/O în C++.

Acest capitol începe cu supraîncărcarea funcţiilor şi se încheie cu studiul supraîncărcării operatorilor. Chiar dacă este similară supraîncărcării funcţiilor, supraîncărcarea operatorilor introduce mai multe nuanţe ale procesului. De aceea, înainte de a încerca să supraîncărcaţi un operator, trebuie să înţelegeţi pe deplin supraîncărcarea funcţiilor.

Supraîncărcarea funcţiilor

După cum s-a discutat în **Capitolul 11**, supraîncărcarea funcţiilor este pur şi simplu procesul de folosire a aceluiaşi nume pentru două sau mai multe funcţii. Punctul esenţial este, totuşi, acela că fiecare redefinire a funcţiei trebuie să folosească sau tipuri diferite, sau un număr diferit de parametri. Compilatorul nu ştie ce funcţie să apeleze, într-o anumită situaţie, decât prin intermediul acestor diferenţe. De exemplu, următorul program supraîncarcă **functiamea()** folosind tipuri diferite de parametri.

```
#include <iostream.h>

int functiamea(int i); /* acestea difera prin tipurile de
                          parametri */
double functiamea(double i);

main()
{
    cout << functiamea(10) << " "; // apeleaza functiamea(int i);
    cout << functiamea(5.4); // apeleaza functiamea(double i)

    return 0;
}

double functiamea(double i)
{
    return i;
}

int functiamea(int i)
{
    return i;
}
```


Următorul program redefineşte **functiamea()** folosind numere diferite de parametri.

```
#include <iostream.h>

int functiamea(int i); /* acestea difera prin numarul de
                          parametri */
int functiamea(int i, int j);

main()
{
    cout << functiamea(10) << " "; //apeleaza functiamea(int i);
    cout << functiamea(4, 5); //apeleaza functiamea(int i, int j)

    return 0;
}

double functiamea(int i)
{
    return i;
}

int functiamea(int i, int j)
{
    return i*j;
}
```

După cum s-a menţionat, punctul cheie al supraîncărcării funcţiilor este acela că funcţiile trebuie să difere prin tipul sau numărul de parametri. Nu pot fi supraîncărcate două funcţii care diferă doar prin tipul returnat. De exemplu, aceasta este o încercare ilegală de a supraîncărca **functiamea()**:

```
int functia mea(int i);
float functiamea(int i);
/* Eroare: tipurile diferite de returnare sunt insuficiente
   pentru supraîncarcare. */
```

Două declaraţii de funcţii par uneori că diferă, dar, de fapt, nu este aşa. De exemplu, să luăm următoarele declaraţii:

```
void f(int *p);
void f(int p[]); // eroare, *p este acelasi cu p[]
```

Amintiți-vă că pentru compilator *p este același lucru cu p[ ]. De aceea, chiar dacă cele două prototipuri par să difere ca tip sau ca parametri, de fapt, nu este așa.

Supraîncărcări de funcții și ambiguități

Puteți crea cazuri în care compilatorul nu este capabil să aleagă între două (sau mai multe) funcții supraîncărcate. Când apare această situație, ea este numită *ambiguă*. Instrucțiunile ambigue sunt erori, iar programul care conține o ambiguitate nu este compilat.

Cauza principală a apariției ambiguităților este, în cele mai multe cazuri, conversia automată a tipului în C++. După cum știți, C++ încearcă să convertească automat argumentele care sunt folosite pentru a apela o funcție în tipurile argumentelor așteptate de funcție. Să luăm, de exemplu, acest fragment:

```
int functiamea(double d);
.
.
.
cout << functiamea('c'); // nu este o eroare, se aplica conversia
```

După cum indică și comentariul, nu este o eroare, deoarece C++ convertește automat caracterul **c** în echivalentul său **double**. În C++ există puține conversii de tip de acest fel care nu sunt admise. Deși conversiile automate sunt foarte utile, ele constituie încă principala sursă a ambiguităților. De exemplu, să luăm următorul program:

```
#include <iostream.h>

float functiamea(float i);
double functiamea(double i);

main()
{
   cout << functiamea(10.1) << " "; /* neambiguu, apeleaza
                                       functiamea(double)*/
   cout << functiamea(10); // ambiguu
   return 0;
}

float functiamea(float i)
{
   return i;
}
double functiamea(double i)
{
   return -i;
}
```


Aici **functiamea()** este supraîncărcată astfel încât să poată prelua argumente ori de tipul **float** ori de tipul **double**. În linia fără ambiguități, este apelată **functiamea(double)** deoarece, doar dacă nu sunt specificate explicit ca fiind **float**, toate constantele în virgulă mobilă din C++ sunt automat de tipul **double**. De aceea, această apelare nu este ambiguă. Dar, când **functiamea()** este apelată folosind numărul întreg 10, se introduce o ambiguitate deoarece compilatorul nu are de unde să știe dacă trebuie convertit în **float** sau în **double**. Aceasta va determina afișarea unui mesaj de eroare, iar programul nu va fi compilat.

După cum ilustrează exemplul precedent, nu supraîncărcarea funcției **functiamea()** relativ la **double** și la **float** este cea care determină ambiguitatea, ci faptul că este apelată cu un argument de tip nedeterminat. Altfel spus, eroarea nu este determinată de supraîncărcarea funcției **functiamea()**, ci de apelarea efectivă.

Iată un alt exemplu de ambiguitate determinată de conversia automată a tipului în C++:

```
#include <iostream.h>

char functiamea(unsigned char ch);

char functiamea(char ch);

main()
{
   cout << functiamea('c'); // aceasta apeleaza functiamea(char)
   cout << functiamea(80) << " "; // ambigua

   return 0;
}

char functiamea(unsigned char ch)
{
   return ch-1;
}

char functiamea(char ch)
{
   return ch+1;
}
```

În C++, **unsigned char** şi **char** *nu* sunt implicit ambigue. Totuşi, când este apelată **functiamea()** folosind numărul întreg 88, compilatorul nu ştie ce funcţie să apeleze. Adică, 88 trebuie convertit în **char** sau în **unsigned char**?

O altă cale prin care puteţi crea ambiguităţi este folosirea argumentelor implicite în funcţiile supraîncărcate. (Argumentele implicite sunt prezentate în **Capitolul 21**.) Pentru a vedea cum, să examinăm următorul program:

```
#include <iostream.h>

int functiamea(int i);

int functiamea(int i, int j=1);

main()
{
    cout << functiamea(4, 5) << " "; // neambigua
    cout << functiamea(10); // ambigua

    return 0;
}

int functiamea(int i)
{
    return i;
}
int functiamea(int i, int j)
{
    return i*j;
}
```

Aici, la prima apelare pentru **functiamea()**, sunt specificate două argumente; de aceea, nu apare nici o ambiguitate, şi este apelată **functiamea(int i, int j)**. Dar, când este apelată a doua oară **functiamea()**, apare ambiguitatea, deoarece compilatorul nu ştie dacă să apeleze versiunea pentru **functiamea()** care are un argument, sau să aplice parametrul implicit versiunii care preia două argumente.

Unele tipuri de supraîncărcare a funcţiilor sunt implicit ambigue chiar dacă, la început, ele nu par astfel. Să considerăm, de exemplu, acest program:

```
// Acest program contine o eroare.
#include <iostream.h>

void f(int x);
void f(int &x); // eroare

main()
{
    int a=10;

    f(a); // eroare, care f()?

    return 0;
}

void f(int x)
{
    cout << "In f(int)\n";
}

void f(int &x)
{
    cout << "In f(int &)\n";
}
```

După cum s-a comentat în programul descris, două funcţii nu pot fi supraîncărcate când singura diferenţă dintre ele este aceea că una preia un parametru referinţă şi alta unul normal, apelat prin valoare. În această situaţie, compilatorul nu are cum să ştie de care versiune este vorba la apelare. Amintiţi-vă că nu este nici o diferenţă între felul în care este specificat un argument când este primit de un parametru de referinţă sau de unul de tip valoare.

Anacronisme pentru supraîncărcare

Când a fost creat C++, pentru a crea o funcţie supraîncărcată a fost necesar cuvântul cheie **overload**. Chiar dacă nu mai este necesar şi este considerat caduc, el este acceptat uzual de compilatoarele de C++ din motive de compatibilitate cu vechile programe în C++. (Totuşi, nu este nici o cerinţă care să garanteze aceasta.) Deoarece puteţi întâlni programe vechi, sau poate că vă aflaţi în situaţia în care este disponibil doar un compilator vechi de C++, este bine să ştiţi cum era folosit **overload**. Iată forma sa generală:

overload: *nume-func*;

Aici, *nume-func* este numele funcţiei pe care o veţi supraîncărca. Acest element trebuie să preceadă declaraţiile de supraîncărcare. De exemplu, următoarea linie spune unui compilator de tip vechi că veţi supraîncărca o funcţie numită **test()**:

```
overload test;
```

Deoarece **overload** este un anacronism, ar trebui să evitaţi utilizarea sa în programele de C++ pe care le creaţi.

Supraîncărcarea funcţiilor constructor

În afară de rolul special de iniţializare, funcţiile constructor nu diferă de alte tipuri de funcţii. Aceasta include şi supraîncărcarea. De fapt, funcţiile constructor supraîncărcate se întâlnesc foarte des. De exemplu, să considerăm următorul program, care creează o **class** numită **data**, care conţine o dată calendaristică. Notaţi supraîncărcarea constructorului în două moduri.

```
#include <iostream.h>
#include <stdio.h>

class data {
    int zi, luna, an;
public:
    data(char *d);
    data(int z, int l, int a);
    void arata_data();
};

// Initializeaza sirul folosit.
data::data(char *d)
{
    sscanf(d, "%d%*c%d%*c%d", &zi, &luna, &an);
}

// Initializeaza intregii folositi.
data::data(int z, int l, int a)
{
    zi = z;
    luna = l;
    an = a;
}
void data::arata_data()
{
    cout << zi << "/" << luna;
    cout << "/" << anul << "\n";
}
main()
```



```
{
    data ob1(4, 12, 96), ob2("22/10/97");

    ob1.arata_data();
    ob2.arata_data();

    return 0;
}
```

În acest program puteţi să iniţializaţi un obiect de tipul **data** specificând data prin trei cifre pentru zi, lună şi an, sau printr-un şir care conţine data în forma sa generală:

zz/ ll /aa

Cel mai uzual motiv pentru a supraîncărca un constructor este de a permite unui obiect să fie creat folosind cele mai convenabile şi mai naturale metode pentru fiecare circumstanţă. De exemplu, în următoarea funcţie **main()** utilizatorului i se solicită data, care este introdusă în matricea **s**. Acest şir poate fi apoi folosit direct pentru a-l crea pe **d**. Nu este necesar ca el să fie convertit în nici o altă formă. Dar, dacă **data()** nu ar fi supraîncărcată pentru a accepta forma de şir, ar fi trebuit să o convertiţi manual în trei întregi de fiecare dată când creaţi un obiect.

```
main()
{
    char s[80];

    cout << "Introduceti noua data: ";
    cin >> s;

    data d(s);
    d.arata_data();

    return 0;
}
```

În altă situaţie poate fi mai convenabilă iniţializarea unui obiect de tip **data** folosind trei întregi. De exemplu, dacă data este generată printr-o metodă de calcul, atunci cel mai natural şi mai corespunzător constructor ce trebuie utilizat este crearea unui obiect **data** folosind **data(int, int, int)**. Cheia este aceea că, aici, prin supraîncărcarea constructorului pentru **data**, l-aţi făcut mai flexibil şi mai uşor de utilizat. Această creştere de flexibilitate şi de uşurinţă în utilizare sunt importante în special dacă doriţi să creaţi biblioteci de clase care vor fi utilizate de alţi programatori.



NOTĂ: C++ defineşte un tip special de constructor supraîncărcat, numit un constructor de copie, care vă permite să determinaţi cum sunt copiate obiectele în anumite circumstanţe. Constructorii de copie vor fi discutaţi mai departe în această carte.

Găsirea adresei unei funcţii supraîncărcate

După cum ştiţi, în C puteţi să atribuiţi adresa unei funcţii unui pointer şi apoi să apelaţi acea funcţie folosind pointerul. Aceeaşi facilitate există, de asemenea, şi în C++. Dar, datorită supraîncărcării funcţiilor, acest proces este ceva mai complex. Pentru a înţelege de ce, să luăm următoarea instrucţiune, care atribuie adresa unei funcţii numită **functiamea()** unui pointer numit **p**.

```
p = functiamea;
```

Dacă aceasta face parte din C, atunci există o funcţie, şi numai una, numită **functiamea()**, iar compilatorul nu are dificultăţi în a-i atribui adresa sa pointerului **p**. Însă, dacă această instrucţiune face parte din C++, atunci **functiamea()** poate fi supraîncărcată. Presupunând că este aşa, cum va şti compilatorul adresa cărei funcţii să o atribuie pointerului **p**? Răspunsul este că totul depinde de felul în care este declarat **p**. De exemplu, să considerăm acest program:

```
#include <iostream.h>

int functiamea(int a);
int functiamea(int a, b);

main()
{
    int (*fp)(int a); // pointer catre int xxx(int)

    fp = functiamea; // indica spre functiamea(int)

    cout << fp(5);

    return 0;
}

int functiamea(int a)
{
    return a;
}

int functiamea(int a, int b)
{
    return a*b;
}
```

Aşa cum ilustrează programul, **fp** este declarat ca un pointer către o funcţie care returnează un întreg şi care preia un argument de tip întreg. C++ foloseşte această informaţie pentru a selecta versiunea **functiamea(int a)** pentru **functiamea()**.

Dacă **fp** ar fi declarat astfel,

```
int (*fp)(int a, int b);
```

atunci lui i s-ar fi atribuit adresa versiunii **functiamea(int a, int b)**, funcţiei **functiamea()**.

Să recapitulăm: când atribuiţi adresa unei funcţii supraîncărcate unui pointer către o funcţie, cea care determină despre care funcţie este vorba este declararea pointerului. Mai mult, declararea funcţiei de tip pointer trebuie să corespundă exact uneia şi numai uneia dintre declarările funcţiei redefinite.

Supraîncărcarea operatorilor

Strâns legată de supraîncărcarea funcţiilor este supraîncărcarea operatorilor. În C++ puteţi să supraîncărcaţi aproape toţi operatorii, astfel încât ei să efectueze operaţii speciale relativ la clasele pe care le creaţi. De exemplu, o clasă care întreţine o memorie stivă poate să supraîncarce + pentru a efectua o operaţie de încărcare, iar -- pentru una de extragere. Când un operator este supraîncărcat, nu se pierde nici una din semnificaţiile sale originale. Se extinde, în schimb, tipul obiectelor cărora li se poate aplica.

Operatorii se supraîncarcă prin crearea funcţiilor **operator**. O funcţie **operator** defineşte operaţiile specifice pe care le va efectua operatorul supraîncărcat relativ la clasa în care este destinat să lucreze. Funcţiile **operator** pot fi sau nu membri ai clasei în care vor opera. Funcţiile **operator** care nu sunt de tip membru sunt, totuşi, de obicei, funcţii **friend** ale clasei. Felul în care sunt scrise funcţiile **operator** diferă pentru cele de tip membru de cele pentru **friend**. De aceea, fiecare categorie va fi examinată separat, începând cu funcţiile **operator** membre.

Crearea unei funcţii operator membru

Funcţiile **operator** membru au această formă generală:

tip-ret nume-clasa::operator#(*lista-arg*)

```
{
    // operaţii
}
```

Deseori funcţiile operator returnează un obiect din clasa asupra căreia operează, dar *tip-ret* poate fi de orice tip valid. # este o rezervare de loc. Când creaţi o funcţie **operator**, înlocuiţi # cu operatorul. De exemplu, dacă supraîncărcaţi operatorul /, folosiţi **operator/**. Când supraîncărcaţi un operator unar, *lista-arg* va fi validă. Când supraîncărcaţi un operator binar, *lista-arg* va conţine un singur parametru. (Motivul acestei situaţii neuzuale va fi clarificat imediat.)

În continuare, este prezentat un exemplu simplu de supraîncărcare de operator. Acest program creează o **class** numită **loc** şi stochează valori de latitudine şi longitudine. El supraîncarcă operatorul + relativ la această clasă. Examinaţi cu atenţie programul, acordând o atenţie specială definirii lui **operator+()**.

```
#include <iostream.h>

class loc {
    int longitud, latitud;
public:
    loc() {}
    loc(int lg, int lt) {
          longitud = lg;
          latitud = lt;
    }

    void arata() {
          cout << longitud << " ";
          cout << latitud << "\n";
    }
    loc operator+(loc op2);
};

loc loc::operator+(loc op2)
{
    loc temp;

    temp.longitud = op2.longitud + longitud;
    temp.latitud = op2.latitud + latitud;

    return temp;
}
```



```
main()
{
    loc ob1(10, 20), ob2(5, 30);

    ob1.arata(); // afiseaza 10 20
    ob2.arata(); // afiseaza 5 30

    ob1 = ob1 + ob2;
    ob1.arata(); // afiseaza 15 50

    return 0;
}
```

După cum puteţi vedea, **operator+()** are doar un parametru, chiar dacă el supraîncarcă operatorul binar +. (Poate că v-aţi aşteptat la doi parametri care să corespundă celor doi operanzi ai operatorului binar.) Motivul pentru care **operator+()** are doar un parametru este acela că operandul din stânga lui + este pasat implicit funcţiei folosind pointerul **this**. Operandul din dreapta este pasat în parametrul **ob2**. Faptul că operandul din stânga este pasat folosind **this** implică, de asemenea, un lucru important: când există operatori binari supraîncărcaţi, cel care generează apelarea funcţiei **operator** este obiectul din stânga.

Cum am mai menţionat, este obişnuit pentru o funcţie **operator** supraîncărcată să returneze un obiect din clasa asupra căreia operează. Făcând aceasta, permiteţi operatorului să fie folosit în expresii C++ mai mari. De exemplu, dacă funcţia **operator+()** returnează alt tip, această expresie nu ar mai fi validă:

```
ob1 = ob1 + ob2;
```

Pentru ca suma lui **ob1** şi **ob2** să fie atribuită lui **ob1**, rezultatul acestei operaţii trebuie să fie un obiect de tip **loc**.

Mai mult, dacă **operator+()** returnează un obiect de tip **loc**, este posibilă următoarea instrucţiune:

```
(ob1 + ob2).arata(); // afiseaza rezultatul lui ob1+ob2
```

În această situaţie, **ob1+ob2** generează un obiect temporar care încetează să mai existe după ce se termină apelarea pentru **arata()**.

Este important să înţelegeţi că o funcţie **operator** poate returna orice tip şi că tipul returnat depinde doar de aplicaţia dvs. efectivă. Numai că, deseori, o funcţie **operator** va returna un obiect de clasa asupra căreia operează.

O ultimă problemă relativ la funcţia **operator()**: ea nu modifică nici unul dintre operanzi. Deoarece utilizarea obişnuită a operatorului + nu modifică nici un

operand, este normal ca nici versiunea supraîncărcată să nu o facă. (De exemplu, 5+7 face 12, dar nici 5 şi nici 7 nu se modifică.) Tot aşa, în urma aplicării asupra unor obiecte din clasa **loc**, **operator+()** nu ar trebui să modifice nici un operand. Chiar dacă sunteţi liber să efectuaţi orice operaţie doriţi în interiorul funcţiei **operator**, în general cel mai bine este să vă menţineţi în contextul utilizării normale a operatorului.

Următorul program adaugă clasei **loc** trei operatori supraîncărcaţi: -, = şi ++ unar. Fiţi foarte atenţi la modul de definire a acestor funcţii.

```
#include <iostream.h>

class loc {
    int longitud, latitud;
public:
    loc() {} // cerut pentru constructii temporare
    loc(int lg, int lt) {
          longitud = lg;
          latitud = lt;
    }

    void arata() {
          cout << longitud << " ":
          cout << latitud << "\n";
    }

    loc operator+(loc op2);
    loc operator-(loc op2);
    loc operator=(loc op2);
    loc operator++();
};
loc loc::operator+(loc op2)
{
    loc temp;
    temp.longitud = op2.longitud + longitud;
    temp.latitud = op2.latitud + latitud;
    return temp;
}
loc loc::operator-(loc op2)
{
    loc temp;
    // observati ordinea operanzilor
    temp.longitud = longitud - op2.longitud;
    temp.latitud = latitud - op2.latitude;
```



```
    return temp;
}
loc loc::operator=(loc op2)
{
    longitud = op2.longitud;
    latitud = op2.latitud;

    return *this; // returneaza obiectul care a generat apelarea
}

loc::operator++()
{
    longitud++;
    latitud++;

    return *this;
}

main()
{
    loc ob1(10, 20), ob2(5, 30), ob3(90, 90);

    ob1.arata();
    ob2.arata();

    ++ob1;
    ob1.arata(); // afiseaza 11 21

    ob2 = ++ob1;
    ob1.arata(); // afiseaza 12 22
    ob2.arata(); // afiseaza 12 22

    ob1 = ob2 = ob3; // atribuire multipla
    ob1.arata(); // afiseaza 90 90
    ob2.arata(); // afiseaza 90 90

    return 0;
}
```

Să studiem pentru început funcţia **operator-()**. Reţineţi ordinea operanzilor pentru scădere. Păstrând semnificaţia scăderii, operandul din dreapta semnului minus este scăzut din operandul din stânga. Deoarece cel care generează apelarea funcţiei **operator-()** este obiectul din stânga, datele din **ob2** trebuie scăzute din

cele spre care indică **this**. Este important să vă reamintiți care operand generează apelarea funcției.

În C++, dacă = nu este supraîncărcat, pentru fiecare clasă pe care o definiți se creează automat o operație implicită de atribuire. Atribuirea implicită este o simplă copiere membru cu membru. Totuși, supraîncărcând =, puteți stabili explicit ce atribuire va face pentru o anumită clasă. În acest exemplu, = supraîncărcat face exact același lucru ca în varianta inițială, dar în alte situații el poate efectua alte operații. Rețineți că funcția **operator=()** returnează ***this**, care este obiectul care a generat apelarea. Acest lucru este necesar dacă doriți să folosiți instrucțiunile de atribuire multiplă, așa cum este aceasta:

```
ob1 = ob2 = ob3; // atribuire multipla
```

În sfârșit, priviți definirea lui **operator++()**. După cum puteți vedea, ea nu are parametri. Deoarece ++ este operator unar, operandul său este transmis implicit folosind pointerul **this**.

Rețineți că atât **operator=()** cât și **operator++()** modifică valoarea unui operand. În cazul atribuirii, operandului din stânga (cel care a generat apelarea funcției **operator=()**) îi este atribuită o nouă valoare. În cazul lui ++, operandul este incrementat. Cum am spus mai devreme, chiar dacă sunteți liber să cereți ca aceste funcții să efectueze orice doriți, aproape întotdeauna este mai înțelept ca să rămână consecvente cu semnificațiile lor originale.

Crearea operatorilor de incrementare și de decrementare cu prefix și cu sufix

În programul precedent a fost supraîncărcată doar forma cu prefix a operatorului de incrementare pentru clasa **loc**. În versiunile vechi de C++ nu era posibil să determinați dacă ++ și -- supraîncărcate precedau sau urmau operandul lor. De exemplu, presupunând un obiect numit **O**, aceste două instrucțiuni erau identice:

```
O++;
++O;
```

Însă, versiunile moderne de C++ asigură o cale de determinare dacă un increment sau un decrement este anterior sau posterior operandului lor. Pentru a realiza aceasta, definiți două versiuni ale funcției **operator++()**. Una este definită așa cum este arătat în programul anterior. Cealaltă este declarată astfel:

```
loc operator++(int x);
```

Dacă ++ precede operandul său, este apelată funcția **operator++()**. Dacă ++ urmează operandului, este apelată **operator++(int x)** iar **x** are valoarea 0.

Exemplul precedent poate fi generalizat. Iată formele generale pentru funcțiile operator ++ și -- cu prefix și cu sufix:

```
// Increment cu prefix
tip operator++() {
    // corpul operatorului cu prefix
}

// Increment cu sufix
tip operator++(intx) {
    // corpul operatorului cu sufix
}

//Decrement cu prefix
tip operator--() {
    // corpul operatorului cu prefix
}

//Decrement cu sufix
tip operator--(int x) {
    corpul operatorului cu sufix
}
```

Supraîncărcarea operatorilor prescurtați

Puteți să supraîncărcați oricare dintre operatorii „prescurtați" din C++, cum sunt +=, -= și ceilalți asemănători lor. De exemplu, această funcție supraîncarcă += relativ la **loc**:

```
loc loc::operator+=(loc op2)
{
    longitud = op2.longitud + longitud;
    latitud = op2.latitud + latitud;

    return *this;
}
```

Țineți minte, atunci când supraîncărcați unul dintre acești operatori, că de fapt combinați o atribuire cu o operație de alt tip.

Restricţii la supraîncărcarea operatorilor

Există unele restricţii care se aplică supraîncărcării operatorilor. Nu puteţi modifica precedenţa unui operator. Nu puteţi să modificaţi numărul operanzilor preluaţi de un operator. (Totuşi, puteţi ignora unul.) Funcţiile operator nu pot avea argumente implicite. În sfârşit, următorii operatori nu pot fi supraîncărcaţi:

. :: .* ?

Amintiţi-vă că, practic, puteţi efectua orice în interiorul unei funcţii **operator**. De exemplu, dacă doriţi să supraîncărcaţi operatorul **+** astfel încât el să scrie de zece ori într-un fişier de pe disc **Imi place C++**, puteţi să o faceţi. Totuşi, dacă vă îndepărtaţi mult de semnificaţia implicită a operatorului, riscaţi să vă destructuraţi programul. De exemplu, dacă cineva care citeşte programul dvs. va întâlni o instrucţiune ca **Ob1+Ob2**, se va aştepta să aibă loc o adunare - nu, să zicem, un acces la disc. De aceea, înainte să disociaţi un operator supraîncărcat de înţelesul său implicit, fiţi sigur că aveţi un motiv serios pentru a o face. Un bun exemplu în care disocierea este o reuşită se regăseşte în modul în care C++ supraîncarcă operatorii de I/O **<<** şi **>>**. Deşi operatorii de I/O nu au nici o legătură cu deplasarea biţilor, ei asigură o „cheie" vizuală asupra semnificaţiei lor atât pentru I/O cât şi pentru deplasarea biţilor. Totuşi, în general, când supraîncărcaţi un operator, cel mai bine este să rămâneţi în contextul semnificaţiei sale implicite.

Cu excepţia operatorului **=**, funcţiile **operator** sunt moştenite de orice clasă derivată. Dar, o **class** derivată este liberă să supraîncarce relativ la ea însăşi orice operator (inclusiv cei supraîncărcaţi de clasa de bază).

Supraîncărcarea operatorilor folosind o funcţie friend

Înainte de a privi câteva exemple de operatori mai exotici, cum ar fi **[ ]**, **new** sau **delete**, este bine să facem o mică digresiune în care se examinează supraîncărcarea folosind funcţiile **friend**.

Puteţi supraîncărca un operator relativ la o clasă folosind o funcţie **friend**. Deoarece un prieten nu este membru al clasei, nu are un pointer de tip **this**. De aceea, unei funcţii supraîncărcate de tip **friend operator** i se transmit explicit operanzii. Aceasta înseamnă că un prieten care supraîncarcă un operator binar are doi parametri, iar unul care supraîncarcă un operator unar are un parametru. Când supraîncărcaţi un operator binar, operandul din stânga este pasat în primul parametru iar cel din dreapta în cel de-al doilea parametru al funcţiei **friend operator**.

În următorul program funcţia **operator+()** este declarată ca fiind **friend**:


```
#include <iostream.h>

class loc {
   int longitud, latitud;
public:
   loc() { } // necesara constructia temporara
   loc(int lg, int lt) {
         longitud = lg;
         latitud = lt;
   }

   void arata() {
         cout << longitud << " ";
         cout << latitud << "\n";
   }
   friend loc operator+(loc op1, loc op2); // acum este prieten
   loc operator-(loc op2);
   loc operator=(loc op2);
   loc operator++();
};

// acum, + este supraincarcat folosind functia prieten
loc operator+(loc op1, loc op2)
{
   loc temp;

   temp.longitud = op1.longitud + op2.longitud;
   temp.latitud = op1.latitud + op2.latitud;

   return temp;
}
loc loc::operator-(loc op2)
{
   loc temp;
   // observati ordinea operanzilor
   temp.longitud = longitud - op2.longitud;
   temp.latitud = latitud - op2.latitud;

   return temp;
}
loc loc::operator = (loc op2)
{
   longitud = op2.longitud;
   latitud = op2.latitud;
```

```
    return *this; // returneaza obiectul care a generat apelarea
loc loc::operator++()
{
    longitud++;
    latitud++;

    return *this;
}

main()
{
    loc ob1(10, 20); ob2(5, 30);

    ob1 = ob2 + ob2;
    ob1.arata();

    return 0;
}
```

Există câteva restricţii care se aplică funcţiilor de tip **friend operator.** Prima, nu puteţi supraîncărca operatorii =, (), [ ] sau -> folosind aceste funcţii. A doua, aşa cum se explică în următorul paragraf, când supraîncărcaţi operatorii de incrementare sau decrementare utilizând funcţii **friend**, trebuie să folosiţi un parametru de referinţă.

Folosirea unui friend pentru a supraîncărca ++ sau --

Dacă doriţi să folosiţi o funcţie **friend** pentru a supraîncărca operatorul de incrementare sau de decrementare, trebuie să transmiteţi operandul ca parametru de referinţă, deoarece funcţiile **friend** nu au pointeri **this**. Presupunând că rămâneţi fideli semnificaţiei originale a operatorilor ++ şi --, aceste operaţii implică modificarea operandului asupra căruia lucrează. Dar, dacă supraîncărcaţi aceste operaţii folosind un **friend**, atunci operandul este transmis ca parametru prin valoare. Aceasta înseamnă că o funcţie de tip **friend operator** nu are cum să modifice operandul. Deoarece acestei funcţii nu i se transmite pointerul **this** către operand, ci doar o copie a acestuia, nici o modificare adusă parametrului nu afectează operandul care generează apelarea. Totuşi, puteţi corecta acest lucru specificând parametrul funcţiei **friend operator** ca referinţă. Acest lucru face ca orice modificări aduse parametrului în interiorul funcţiei să afecteze operandul care a generat apelarea. De exemplu, următorul program foloseşte funcţiile **friend** pentru a supraîncărca versiunile cu prefix ale operatorilor ++ şi -- relativ la clasa **loc**.


```
#include iostream.h>

class loc {
   int longitud, latitud;
public:
   loc() { }
   loc(int lg, int lt) {
         longitud = lg;
         latitud = lt;
   }

   void arata() {
         cout << longitud << " ";
         cout << latitud << "\n";
   }

   loc operator=(loc op2);
   friend loc operator++(loc &op);
   friend loc operator--(loc &op);
};

loc loc::operator=(loc op2)
{
   longitud = op2.longitud
   latitud = op2.latitud;
   return *this; // returneaza obiectul care a generat apelarea
}
// acum un prieten - foloseste referinta
loc operator++(loc &op)
{
   op.longitud++;
   op.latitude++;

   return op;
}
// il face prieten pe op -- - foloseste referinta
loc operator--(loc &op)
{
   op.longitud--;
   op.latitud--;

   return op;
}
```

```
main()
{
    loc ob1(10, 20), ob2;

    ob1.arată();
    ++ob1;
    ob1.arata(); // afiseaza 11 21

    ob2 = ++ob1;
    ob1.arata(); afiseaza 12 22

    --ob2;
    ob2.arata(); // afiseaza 11 21

    return 0;
}
```

Dacă doriţi să supraîncărcaţi versiunea cu sufix a operatorilor de incrementare şi de decrementare folosind **friend**, specificaţi simplu un al doilea parametru întreg, fictiv. De exemplu, aceste linii arată prototipul versiunii **friend** cu sufix pentru operatorul de incrementare relativ la **loc**:

```
// versiune prieten cu sufix a lui ++
friend loc operator++(loc &op, int x);
```

Funcţiile friend operator adaugă flexibilitate

În multe cazuri, nu există o diferenţă funcţională între supraîncărcarea unui operator folosind o funcţie **friend** sau una de tip membru. Pentru a păstra cel mai înalt grad de încapsulare, cel mai bine este ca, în aceste cazuri, să supraîncărcaţi folosind funcţiile membre. Totuşi, există o situaţie în care utilizarea unui prieten creşte flexibilitatea unui operator supraîncărcat. Să studiem acum acest caz.

După cum ştiţi, când supraîncărcaţi un operator binar folosind o funcţie membru, obiectul din stânga operatorului este cel ce generează apelarea funcţiei **operator** supraîncărcate. Apoi, în pointerul **this** este transmis un pointer către acest obiect. Acum, presupuneţi o clasă numită **CL**, pentru care adunarea unui obiect cu un întreg este redefinită cu ajutorul unei funcţiei membre **operator**. Având un obiect al acelei clase numit **Ob**, următoarea expresie este validă:

```
Ob +100 // valida
```


În acest caz, **Ob** generează apelarea funcţiei supraîncărcate + şi se efectuează adunarea. Dar, ce se întâmplă dacă expresia este scrisă astfel?

```
100 + Ob // invalida
```

În acest caz, cel care apare în stânga este întregul. Deoarece întregul este un tip încorporat, între acesta şi un tip **Ob** nu este definită nici o operaţie. De aceea, compilatorul nu va compila această expresie. După cum vă puteţi imagina, în unele aplicaţii, să trebuiască mereu ca obiectul să fie poziţionat în stânga este o povară care determină eşecuri.

Soluţia problemei precedente este să supraîncărcaţi adunarea folosind o funcţie **friend**, nu o funcţie membru. În acest caz, funcţiei **operator** îi sunt transmise explicit ambele argumente. De aceea, pentru a permite atât *obiect+întreg* cât şi *întreg+obiect*, supraîncărcaţi pur şi simplu funcţia de două ori - o versiune pentru fiecare situaţie. În acest fel, când supraîncărcaţi un operator folosind două funcţii **friend**, obiectul poate să apară atât în partea stângă cât şi în partea dreaptă a operatorului.

Acest program ilustrează cum sunt folosite funcţiile **friend** pentru a defini o operaţie care implică un obiect şi un tip încorporat.

```
#include <iostream.h>

class loc {
    int longitud, latitud;
public:
    loc() { }
    loc(int lg, int lt) {
        longitud = lg;
        latitud = lt;
    }

    void arata() {
        cout << longitud << " ";
        cout << latitud << "\n";
    }

    loc operator+(loc op2);
    friend loc operator+(loc op1, int op2);
    friend loc operator+(int op1, loc op2);
};

loc loc::operator+(loc op2)
{
```

```
    loc temp;
    temp.longitud = op2.longitud + longitud;
    temp.latitud = op2.latitud + latitud;
    return temp;
}

// + este supraincarcat pentru loc + int
loc operator+(loc op1, int op2)
{
    loc temp;
    temp.longitud = op1.longitud + op2;
    temp.latitud = op1.latitud + op2;
    return temp;
}

// + este supraincarcat pentru int + loc
loc operator+(int op1, loc op2)
{
    loc temp;
    temp.longitud = op1 + op2.longitud;
    temp.latitud = op1 + op2.latitud;
    return temp;
}

main()
{
    loc ob1(10, 20), ob2(5, 30), ob3(7, 14);

    ob1.arata();
    ob2.arata();
    ob3.arata();

    ob1 = ob2 + 10; // amandoua
    ob3 = 10 + ob2; // sunt valide

    ob1.arata();
    ob3.arata;

    return 0;
}
```


Supraîncărcarea operatorilor new şi delete

Este posibil ca **new** şi **delete** să fie supraîncărcate. Puteţi face acest lucru dacă doriţi să folosiţi unele metode speciale de alocare de memorie. De exemplu, puteţi avea nevoie de rutine de alocare care, atunci când s-a epuizat memoria disponibilă (heap), să folosească automat un fişier de pe disc ca memorie virtuală. Indiferent de motiv, este un lucru foarte simplu să supraîncărcaţi aceşti operatori.

Aici sunt prezentate scheletele funcţiilor care redefinesc **new** şi **delete**:

```
void *operator new(size_t marime)
{
    // efectueaza alocarea
    return pointer_la_memorie;
}

void operator delete(void *p)
{
    // memoria libera este indicata de p
}
```

Tipul **size_t** este definit ca un tip apt să conţină cea mai mare porţiune de memorie contiguă care poate fi alocată. **size_t** este de tip întreg fără semn. Parametrul **marime** va conţine numărul de octeţi necesar pentru a memora obiectul pentru care se face alocarea. Funcţia **new** supraîncărcată trebuie să returneze un pointer spre memoria alocată, sau zero, dacă apare o eroare de alocare. Cu excepţia acestor obligaţii, funcţia **new** supraîncărcată poate să facă orice altceva îi cereţi.

Funcţia **delete** primeşte un pointer spre regiunea de memorie pe care trebuie să o elibereze şi eliberează sistemului memoria alocată anterior.

Operatorii **new** şi **delete** pot fi suprapuşi global, astfel încât orice utilizare a lor să apeleze versiunile stabilite de dvs.. De asemenea, ele pot fi supraîncărcate relativ la una sau mai multe clase. Să începem cu un exemplu de supraîncărcare pentru **new** şi **delete** relativ la o clasă. Pentru simplitate, nu va fi folosită o manieră nouă de alocare, ci funcţiile supraîncărcate vor invoca pur şi simplu **malloc()** şi **free()**. (În aplicaţiile dvs. puteţi, desigur, să introduceţi altă metodă, oarecare, de alocare.)

Pentru a supraîncărca operatorii **new** şi **delete** relativ la o clasă, declaraţi, pur şi simplu, funcţiile **operator** supraîncărcate ca membre ale clasei. Iată, de exemplu, operatorii **new** şi **delete** supraîncărcaţi pentru clasa **loc**:

```
#include <iostream.h>
#include <stdlib.h>
```

```
class loc {
   int longitud, latitud;
public:
   loc() {}
   loc(int lg, int lt) {
         longitud = lg;
         latitud = lt;
   }

   void arata() {
         cout << longitud << " ";
         cout << latitud << "\n";
   }

   void *operator new(size_t marime);
   void operator delete(void *p);
};

// new supraincarcat relativ la loc
void *loc::operator new(size_t marime)
{
   cout << "In new al meu\n";
   return malloc(marime);
}

// delete supraincarcat relativ la loc
void loc::operator delete(void *p)
{
   cout << "In delete al meu\n";
   free(p);
}

main()
{
   loc *p1, *p2;
   p1 = new loc (10, 20);
   if(!p1) {
         cout << "Eroare de alocare\n";
         exit(1);
   }

   p2 = new loc (-10, -20);
   if(!p2) {
```



```
         cout << "Eroare de alocare\n";
         exit(1);
   }

   p1->arata();
   p2->arata();

   delete p1;
   delete p2;

   return 0;
}
```

Când **new** şi **delete** sunt supraîncărcaţi relativ la o anumită clasă, utilizarea acestor operatori asupra oricărui alt tip de date determină efectuarea operaţiilor **new** şi **delete** iniţiale. Operatorii suprapuşi se aplică doar acelor tipuri pentru care au fost definiţi. Aceasta înseamnă că, dacă adăugaţi următoarea linie în **main()**, **new** va fi efectuat implicit.

```
int *f = new float; // foloseste new implicit
```

Puteţi redefini global **new** şi **delete** prin supraîncărcarea acestor operatori, în afara oricărei declaraţii de clase. Când aceştia sunt suprapuşi global, sunt ignoraţi **new** şi **delete** impliciţi din C++, iar pentru toate alocările cerute sunt folosiţi noii operatori. Desigur, dacă aţi definit vreo versiune de **new** şi **delete** relativ la una sau mai multe clase, atunci când se alocă memorie obiectelor din clasa pentru care au fost definite, vor fi folosite versiunile specifice acelor clase. Cu alte cuvinte, când sunt întâlnite ori **new** ori **delete**, compilatorul verifică mai întâi dacă au fost definite relativ la clasa asupra cărora operează. Dacă este aşa, vor fi folosite acele versiuni specifice. Dacă nu, C++ foloseşte **new** şi **delete** definite global. Dacă acestea au fost supraîncărcate, atunci sunt utilizate aceste versiuni.

Pentru a vedea un exemplu de supraîncărcare globală pentru **new** şi **delete**, studiaţi acest program:

```
#include <iostream.h>
#include <stdlib.h>

class loc {
   int longitud, latitud;
public:
   loc() {}
   loc(int lg, int lt) {
         longitud = lg;
```

```
            latitud = lt;
    }

    void arata() {
            cout << longitud  << " ";
            cout << latitud << "\n";
    }
};

// new global
void *operator new(size_t marime)
{
    return malloc(marime);
}

// delete global
void operator delete(void *p)
{
    free(p);
}

main()
{
    loc *p1, *p2;
    p1 = new loc (10, 20);
    if(!p1) {
            cout << "Eroare de alocare\n";
            exit(1);
    }
p2 = new loc (-10, 20);
if(!p2) {
    cout << "Eroare de alocare\n";
    exit(1);
}

    float *f = new float;
    // foloseste, de asemenea, new supraincarcat
    if(!f) {
            cout << "Eroare de alocare\n";
            exit(1);
    }

    *f = 10.10;
```



```
    cout <, *f << "\n";

    p1->arata();
    p2->arata();

    delete p1;
    delete p2;
    delete f; // foloseste delete supraincarcat

    return 0;
}
```

Rulaţi acest program pentru a vă dovedi că operatorii **new** şi **delete** încorporaţi au fost, într-adevăr, supraîncărcaţi.

Supraîncărcarea operatorilor new şi delete pentru matrice

Dacă doriţi să puteţi aloca memorie matricelor de obiecte folosind sistemul dvs. propriu de alocare, va trebui să supraîncărcaţi **new** şi **delete** a doua oară. Pentru a aloca şi a elibera memorie pentru matrice, trebuie să folosiţi aceste forme:

```
// Aloca memorie unei matrice de obiecte.
void *operator new[](size_t marime)
{
    // Efectueaza alocarea.
    return pointer_la_memorie;
}

// Delete pentru o matrice de obiecte.
void operator delete[](void *p)
{
    /* Memoria libera este indicata de p.
    Destructor apelat automat pentru fiecare element.
    */
}
```

Când se alocă memorie unei matrice, este apelată automat funcţia constructor a fiecărui obiect al acelei matrice. Când este eliberată memoria, se apelează automat funcţia destructor. Nu este nevoie să asiguraţi un cod explicit pentru a realiza aceste acţiuni.

Următorul program alocă şi eliberează memorie pentru un obiect şi pentru o matrice de obiecte de tipul **loc**.

```
#include <iostream.h>
#include <stdlib.h>

class loc {
    int longitud, latitud;
public:
    loc() {longitud = latitud = 0;}
    loc(int lg, int lt) {
          longitud = lg;
          latitud = lt;
    }

    void arata() {
          cout << longitud << " ";
          cout << latitud << "\n;
    }

    void *operator new(size_t marime);
    void operator delete(void *p);
    void *operator new[](size_t marime);
    void operator delete[](void *p);
};

// new supraincarcat relativ la loc
void *loc::operator new(size_t marime)
{
    cout << "In new al meu\n";
    return malloc(marime);
}

// delete supraincarcat relativ la loc
void loc::operator delete (void *p)"
{
    cout << "In delete al meu\n";
    free(p);
}

// new supraincarcat relativ la loc, pentru matrice
void *loc::operator new[](size_t marime)
{
    cout << "Aloca memorie pentru matrice folosind
              new[] propriu\n";
    return malloc(marime);
```



```
}

// delete supraincarcat relativ la loc, pentru matrice
void loc::operator delete[](void *p)
{
    cout << "Elibereaza memorie din matrice folosind
              delete[] propriu\n";
    free(p);
}

main()
{
    loc *p1, *p2;
    int i;

    p1 = new loc (10, 20); // aloca memorie pentru un obiect
    if(!p1) {
          cout << "Eroare de alocare\n";
          exit(1);
    }

    p2 = new loc [10]; // aloca memorie pentru o matrice
    if(!p2) {
          cout << "Eroare de alocare\n";
          exit(1);
    }

    p1->arata();

    for(i=0; i<10; i++)
          p2[i].arata();

    delete p1; // elibereaza un obiect
    delete [] p2; // elibereaza o matrice

    return 0;
}
```

Supraîncărcarea unor operatori speciali

C++ şi predecesorul său C definesc ca operaţii înscrierea matricelor, apelarea funcţiilor şi „indicare spre". Operatorii care efectuează aceste funcţii sunt [ ], () şi

respectiv ->. Aceşti operatori destul de exotici pot fi supraîncărcaţi în C++, înlesnind câteva utilizări foarte interesante.

Există o restricţie importantă care se aplică acestor trei operatori: ei trebuie să fie funcţii membre care nu sunt de tip **static**; nu pot fi **friend**.

Supraîncărcarea pentru [ ]

În C++, atunci când este supraîncărcat, [ ] este considerat a fi un operator binar. De aceea, forma generală a unei funcţii membru de tip **operator[ ]()** este cea prezentată aici:

```
tip nume-clasa::operator[ ](int i)
{
    // ...
}
```

Practic, nu este necesar ca parametrii să fie de tip **int**, dar o funcţie **operator[]()** este folosită tipic pentru a asigura înscrierea indecşilor într-o matrice şi, de aceea, este folosită, în general, o valoare întreagă.

Dându-se un obiect numit **O**, expresia

```
O[3]
```

se transformă în această apelare a funcţiei **operator[ ]()**:

```
operator[](3)
```

Aceasta înseamnă că valoarea expresiei din operatorul de înscriere este transmisă funcţiei **operator[]()** cu parametrul său explicit. Pointerul **this** va indica spre **O**, obiectul care a generat apelarea.

În următorul program, **untip** declară o matrice formată din trei întregi. Funcţia sa constructor iniţializează fiecare element al matricei cu valoarea specificată. Funcţia **operator[]()** supraîncărcată returnează valoarea matricei, având ca indice valoarea parametrului său.

```
#include <iostream.h>

class untip {
    int a[3];
public:
    untip(int i, int j, int k) {
          a[0] = i;
          a[1] = j;
          a[2] = k;
    }
    int operator[](int i) {return a[i];}
};

main()
{
    untip ob(1, 2, 3);

    cout << ob[1]; // afiseaza 2

    return 0;
}
```

Puteţi proiecta funcţia **operator[]()** astfel încât [ ] să fie folosit atât în partea stângă, cât şi în partea dreaptă a unei instrucţiuni de atribuire. Pentru a face aceasta, specificaţi pur şi simplu valoarea returnată de **operator[]()** ca referinţă. Următorul program face această modificare şi îi arată modul de folosire.

```
#include <iostream.h>

class untip {
    int a[3];
public:
    untip(int i, int j, int k) {
          a[0] = i;
          a[1] = j;
          a[2] = k;
    }
    int &operator[](int i) {return a[i];}
};

main()
{
    untip ob(1, 2, 3);

    cout << ob[1]; // afiseaza 2
    cout << " ";

    ob[1] = 25; // [] in stinga lui =

    cout << ob[1]; // acum afiseaza 25
```

```
    return 0;
}
```

Deoarece **operator[ ]()** returnează acum o referinţă către elementul matricei cu indicele i, el poate fi folosit în partea stângă a unei atribuiri, pentru a modifica un element al matricei. (Desigur, poate fi folosit în continuare la fel de bine şi în partea dreaptă.)

Un avantaj al supraîncărcării operatorului [ ] este acela că el permite o cale de utilizare sigură a indicilor matricelor în C++. După cum ştiţi, este posibil ca, în C++, să depăşiţi în sus sau în jos limitele matricei în timpul rulării fără să primiţi un mesaj de eroare.

Însă, dacă stabiliţi o clasă care conţine matricea şi permiteţi accesul la ea prin operatorul de înscriere supraîncărcat [ ], atunci puteţi intercepta indicii din afara limitei. De exemplu, următorul program adaugă, faţă de precedentul, o verificare a limitei şi dovedeşte că ea funcţionează.

```
// Un exemplu de matrice sigura.
#include <iostream.h>
#include <stdlib.h>

class untip {
    int a[3];
public:
    untip(int i, int j, int k) {
            a[0] = i;
            a[1] = j;
            a[2] = k;
    }
    int &operator[](int i) {return a[i];}
};

// Asigura verificarea limitei pentru untip.
int &untip::operator[](int i)
{
    if(i<0 || i>2) {
            cout << "Eroare de limita\n";
            exit(1);
    }
    return a[i];
}

main()
{
        untip ob(1, 2, 3);
        cout << ob[1]; // afiseaza 2
        cout << " ";
        ob[1] = 25; // [] apare in stanga
        cout << ob[1]; // afiseaza 25

        ob[3] = 44; /* determina eroare in timpul rularii, 3 in
                       afara limitei */

        return 0;
}
```

În acest program, atunci când se execută instrucţiunea

```
ob[3] = 44;
```

eroarea de depăşire a limitelor este depistată de **operator[ ]()**, iar programul se încheie înainte de a se produce vreo pagubă. (În practică, pot fi apelate anumite funcţii de tratare a erorilor care rezolvă condiţiile de ieşire din limite; nu va trebui ca programul să se încheie).

Supraîncărcarea pentru ()

Când supraîncărcaţi operatorul de apelare a funcţiei (), nu creaţi o nouă cale de apelare a unei funcţii. Creaţi, de fapt, o funcţie **operator** căreia îi poate fi transmis un număr arbitrar de parametri. Să începem cu un exemplu. Dându-se declararea funcţiei **operator** supraîncărcate

```
double operator()(int a, float f, char *s);
```

şi un obiect **O** de această **class**, atunci instrucţiunea

```
O(10, 23.34, "hi");
```

este transpusă în următoarea apelare a funcţiei **operator()**:

```
operator()(10, 23.34, "hi");
```

În general, când supraîncărcaţi operatorul (), definiţi parametrii pe care doriţi să îi transmiteţi acelei funcţii. Când folosiţi în programele dvs. operatorul (), argumentele pe care le specificaţi sunt copiate în acei parametri. Ca de obicei, obiectul care generează apelarea (în acest exemplu, **O**) este indicat de pointerul **this**.

Iată un exemplu de supraîncărcare pentru **()** relativ la clasa **loc**. El atribuie obiectului căruia i se aplică valoarea celor două argumente pentru longitudine şi latitudine.

```
#include <iostream.h>

class loc {
    int longitud, latitud;
public:
    loc() { }
    loc(int lg, int lt) {
          longitud = lg;
          latitud = lt;
    }

    void arata() {
          cout << longitud << " ";
          cout << latitud << "\n";
    }
    loc operator+(loc op2);
    loc operator()(int i, int j);
};

loc loc::operator()(int i, int j)
{
    longitud = i;
    latitud = j;

    return *this;
}

loc loc::operator+(loc op2)
{
    loc temp;
    temp.longitud = op2.longitud + longitud;
    temp.latitud = op2.latitud + latitud;

    return temp;
}

main()
{
    loc ob1(10, 20), ob2(1, 1);
```



```
    ob1.arata();
    ob1(7,8); // poate fi executata singura
    ob1.arata();

    ob1 = ob2 + ob1(10, 10); // poate fi folosita in expresii
    ob1.arata();

    return 0;
}
```

REŢINEŢI: *Când supraîncărcaţi (), puteţi folosi orice tip de parametri şi returna orice tip de valoare. Aceste tipuri vor fi dictate de cerinţele programelor dvs.*

Supraîncărcarea pentru ->

Când este supraîncărcat, operatorul pointer -> este considerat un operator unar. Iată modul său de utilizare generală:

obiect->element;

obiect este, aici, obiectul care activează apelarea. Funcţia **operator->()** trebuie să returneze un pointer spre un obiect de clasa asupra căreia operează **operator->()**. *element* trebuie să fie un element accesibil din cadrul obiectului returnat de **operator->()**.

Următorul program ilustrează supraîncărcarea lui **->** arătând echivalenţa dintre **ob.i** şi **ob->i** când **operator->()** returnează pointerul **this**.

```
#include <iostream.h>

class clasamea {
public:
    int i;
    clasamea *operator->() {return this;}
};

main()
{
    clasamea ob;

    ob->i = 10; // identic cu ob.i
```

```
    cout << ob.i << " " << ob->i;

    return 0;
}
```

Supraîncărcarea operatorului virgulă

Puteţi supraîncărca operatorul virgulă. Virgula este un operator binar şi, ca pentru toţi operatorii suprapuşi, puteţi să determinaţi ca o virgulă supraîncărcată să efectueze orice operaţie doriţi. Totuşi, dacă doriţi ca virgula supraîncărcată să acţioneze într-un mod similar cu operaţia sa normală, ea trebuie să renunţe la valoarea din termenul său stâng şi să atribuie valoarea operaţiei termenului din dreapta. Într-o listă separată prin virgulă trebuie să se renunţe la toţi termenii, în afară de cel din extrema dreaptă. După cum ştiţi, acesta este felul în care virgula lucrează implicit în C++.

Iată un program care ilustrează efectul supraîncărcării operatorului virgulă în modul său de operare implicit.

```
#include <iostream.h>

class loc {
    int longitud, latitud;
public:
    loc() { }
    loc(int lg, int lt) {
        longitud = lg;
        latitud = lt;
    }

    void arata() {
        cout << longitud << " ";
        cout << latitud << "\n";
    }
    loc operator+(loc op2);
    loc operator,(loc op2);
};

loc loc::operator,(loc op2)
{
    loc temp;
```



```
    temp.longitud = op2.longitud;
    temp.latitud = op2.latitude;
    cout << op2.longitud << " " << op2.latitude << "\n";

    return temp;
}
loc loc::operator+(loc op2)
{
    loc temp;

    temp.longitud = op2.longitud + longitud;
    temp.latitud = op2.latitud + latitud;

    return temp;
}

main()
{
    loc ob1(10, 20), ob2(5, 30), ob3(1, 1);

    ob1.arata();
    ob2.arata();
    ob3.arata();

    cout << "\n";

    ob1 = (ob1, ob2+ob2, ob3);

    ob1.arata(); // afiseaza 1 1, valoarea lui ob3

    return 0;
}
```

Acest program afişează următoarea ieşire:

```
10 20
5 30
1 1

10 60
1 1
1 1
```

Observați că, deși se renunță la valorile operanzilor din partea stângă, fiecare expresie este executată de către compilator, deci va apărea orice efect secundar dorit.

Mai rețineți că operandul din partea dreaptă este transmis prin **this**, iar valoarea sa este eliminată de către funcția **operator,()**. Funcția returnează valoarea din dreapta operației. Aceasta determină ca virgula supraîncărcată să se comporte similar cu operația sa implicită. Dacă doriți să supraîncărcați virgula pentru a face altceva, va trebui să modificați aceste două caracteristici.

Capitolul 15

Moștenirea

C++

Moştenirea este una dintre pietrele de temelie ale OOP deoarece ea permite crearea clasificărilor ierarhice. Utilizând moştenirea, puteţi să construiţi o clasă generală care defineşte trăsăturile comune ale unui set de elemente corelate. Această clasă poate fi apoi moştenită de alte clase, particulare, fiecare adăugând doar acele elemente care îi sunt proprii.

Pentru a ne menţine în terminologia C++ standard, o clasă care este moştenită este numită o *clasă de bază*. Clasa care moşteneşte este numită *clasă derivată*. Mai departe, o clasă derivată poate fi folosită ca una de bază pentru altă clasă derivată. Astfel se realizează moştenirea multiplă.

Suportul pentru moştenire în C++ este bogat şi flexibil. Moştenirea a fost prezentată în **Capitolul 11**. Aici ea este examinată în detaliu.

Controlul accesului la clasa de bază

După cum ştiţi, când o clasă moşteneşte alta, se foloseşte următoarea formă generală:

class *nume-clasa-derivata* : *acces nume-clasa-de-baza* {
    // corpul clasei
};

Când o clasă moşteneşte alta, membrii clasei de bază devin membri ai clasei derivate. Accesul la membrii clasei din interiorul clasei derivate este determinat prin *acces*. Specificatorul de acces al clasei de bază trebuie să fie **public**, **private** sau **protected**. În cazul în care nu este prezent nici un specificator, el este implicit **private** pentru o clasă derivată de tip **class**. Dacă clasa derivată este o **struct**, atunci, în absenţa unui specificator explicit, accesul este implicit **public**. Să examinăm ramificaţiile utilizării accesului **public** sau **private**. (Specificatorul **protected** este studiat în paragraful următor.)

Când specificatorul de acces al unei clase de bază este **public**, toţi membrii publici ai clasei devin membri publici ai clasei derivate, iar toţi membrii **protected** (protejaţi) ai bazei devin membri protejaţi ai clasei derivate. În toate cazurile, elementele **private** (particulare) ale bazei rămân particulare pentru bază şi nu sunt accesibile membrilor clasei derivate. De exemplu, aşa cum se ilustrează în următorul program, obiectele de tip **derivat** pot fi accesibile direct membrilor publici din **baza**.

```
#include <iostream.h>

class baza {
   int i, j;
public:
   void pune(int a, int b) {i=a; j=b;}
```



```
   void arata() { cout << i << " " << j << "\n";}
};

class derivat : public baza {
   int k;
public:
   derivat(int x) {k=x;}
   void aratak() { cout << k << "\n";}
};

main()
{
   derivat ob(3);
   ob.pune(1, 2); // acces la membrul bazei
   ob.arata(); // acces la membrul bazei

   ob.aratak(); // foloseste membrul clasei derivate

   return 0;
}
```

Când clasa de bază este moştenită prin utilizarea specificatorului de acces **private**, toţi membrii publici şi protejaţi ai clasei de bază devin membri privaţi ai clasei derivate. De exemplu, următorul program nici măcar nu va fi compilat deoarece atât **pune()**, cât şi **arata()** sunt acum elemente particulare pentru **derivat**.

```
// Acest program nu va fi compilat
#include <iostream.h>

class baza {
   int i, j;
public:
   void pune(int a, int b) {i=a; j=b;}
   void arata() { cout << i << " " << j << "\n";}
};

// Elementele publice din baza sunt particulare in derivat.
class derivat : private baza {
   int k;
public:
   derivat(int x) {k=x;}
   void aratak() {cout << k << "\n";
};
```

```
main()
{
    derivat ob(3);

    ob.pune(1, 2); // eroare, nu poate avea acces la pune()
    ob.arata(); // eroare, nu poate avea acces la arata()

    return 0;
}
```



*REȚINEȚI: Când un specificator de acces al unei clase de bază este **private**, membrii publici sau protejați ai bazei devin membri particulari ai clasei derivate. Aceasta înseamnă că ei sunt încă accesibili membrilor clasei derivate, dar și secțiunilor din programul dvs. care nu sunt membri nici ai clasei de bază, nici ai celei derivate.*

Moștenirea și membrii protejați

Cuvântul cheie **protected** este inclus în C++ pentru a oferi o flexibilitate mai mare mecanismului de moștenire. Când un membru al clasei este declarat ca fiind **protected**, acel membru nu este accesibil altor elemente ale programului care nu sunt membri ai clasei. Cu o singură excepție importantă, accesul la un membru protejat este același ca și accesul la un membru particular - el este accesibil doar altor membri din aceeași clasă cu a sa. Singura excepție este atunci când membrul protejat este moștenit. În acest caz, un membru protejat diferă esențial de unul particular.

După cum știți din paragraful anterior, un membru particular al unei clase de bază nu este accesibil altei secțiuni ale programului dvs., inclusiv orice clasă derivată. Dar, membrii **protected** se comportă diferit. În cazul în care clasa de bază este moștenită ca **public**, atunci membrii protejați ai acestei clase devin membri protejați ai clasei derivate și, de aceea, accesibili acesteia. Cu alte cuvinte, folosind **protected**, puteți crea membri ai clasei care sunt particulari în clasa lor, dar care pot fi moșteniți și sunt accesibili unei clase derivate. Iată un exemplu:

```
#include <iostream.h>

class baza {
protected:
    int i, j; /* particulari pentru baza, dar accesibili
                 pentru derivat */
```



```
public:
    void pune(int a, int b) {i=a; j=b;}
    void arata() { cout << i << " " << j << "\n";}
};

class derivat : public baza {
    int k;
public:
    // derivat poate sa aiba acces la i si j din baza
    void punek() {k=i*j;}

    void aratak() {cout << k << "\n";}
};

main()
{
    derivat ob;

    ob.pune(2, 3); // OK, cunoscut lui derivat
    ob.arata(); // OK, cunoscut lui derivat

    ob.punek();
    ob.aratak();

    return 0;
}
```

Aici, deoarece **baza** este moștenită de **derivat** ca **public**, iar **i** și **j** sunt declarate ca fiind **protected**, funcția **punek()** din **derivat** poate să aibă acces la ele. Dacă **i** și **j** ar fi fost declarate ca fiind **private** în **baza**, atunci **derivat** nu ar fi avut acces la ele, iar programul nu ar fi fost compilat.

Când o clasă derivată este folosită ca o clasă de bază pentru altă clasă derivată, atunci orice membru protejat al clasei de bază inițiale care este moștenită (ca **public**) de către prima clasă derivată poate fi moștenit, de asemenea, ca **protected** și de cea de-a doua clasă derivată. De exemplu, următorul program este corect, iar **derivat2** poate, într-adevăr, să aibă acces la **i** și **j**.

```
#include <iostream.h>

class baza {
protected:
    int i, j;
```

```
public:
    void pune(int a, int b) {i=a; j=b;}
    void arata() { cout << i << " " << j << "\n";}
};

// i si j mosteniti ca protejati.
class derivat1 : public baza {
    int k;
public:
    void punek() {k=i*j;} // legal
    void aratak() {cout << k << "\n";}
};

// i si j mosteniti indirect prin derivat1.
class derivat2 : public derivat1 {
    int m;
public:
    void punem() {m = i-j;} // posibil
    void aratam() {cout << m << "\n";}
};

main()
{
    derivat1 ob1;
    derivat2 ob2;

    ob1.pune(2, 3);
    ob1.arata();
    ob1.punek();
    ob1.aratak();

    ob2.pune(3, 4);
    ob2.arata();
    ob2.punek();
    ob2.punem();
    ob2.aratak();
    ob2.aratam();

    return 0;
}
```

Dacă, totuşi, **baza** ar fi moştenită ca fiind **private**, atunci toţi membrii din **baza** ar deveni membri **private** ai lui **derivat1**, ceea ce înseamnă că ei nu ar fi accesibili

lui **derivat2**. (Totuşi, **i** şi **j** ar continua să fie accesibili lui **derivat1**.) Această situaţie este ilustrată de către următorul program, care conţine erori (şi nu va fi compilat). Comentariile descriu fiecare eroare.

```
// Acest program nu va fi compilat.
#include <iostream.h>

class baza {
protected:
    int i, j;
public:
    void pune(int a, int b) {i=a; j=b;}
    void arata() { cout << i << " " << j << "\n";}
};

// Acum, toate elementele din baza sunt particulare in derivat1.
class derivat1 : private baza {
    int k;
public:
    // ilegal deoarece i si j sunt particulari in derivat1
    void punek() {k=i*j;} // OK
    void aratak() {cout << k << "\n";}
};

// Accesul la i, j, pune() si arata() nu este mostenit.
class derivat2 : public derivat1 {
    int m;
public:
    // ilegal deoarece i si j sunt particulari in derivat1
    void punem() {m = i-j;} // eroare
    void aratam() {cout << m << "\n";}
};

main()
{
    derivat1 ob1;
    derivat2 ob2;

    ob1.pune(1, 2); // eroare, nu poate folosi pune()
    ob1.arata(); // eroare, nu poate folosi arata()

    ob2.pune(3, 4);  // eroare, nu poate folosi pune()
    ob2.arata(); // eroare, nu poate folosi arata()
```

```
    return 0;
}
```

NOTĂ: *Chiar dacă **baza** este moştenită ca fiind **private** în **derivat1**, **derivat1** are încă acces la elementele **public** şi **protected** din **baza**. Totuşi, el nu poate transmite mai departe acest privilegiu.*

Moştenirea protected a clasei de bază

Este posibil să moşteniţi o clasă de bază ca **protected**. Când se face acest lucru, toţi membrii publici şi protejaţi ai clasei de bază devin membri protejaţi ai clasei derivate. Iată un exemplu:

```
#include <iostream.h>

class baza {
protected:
    int i, j; /* particulari pentru baza, dar accesibili
                 pentru derivat */
public:
    void puneij(int a, int b) {i=a; j=b;}
    void arataij() { cout << i << " " << j << "\n";}
};

// Mosteneste baza ca protected.
class derivat1 : protected baza {
    int k;
public:
    /* derivat poate avea acces la i si j din baza si la
       puneij(). */
    void punek() {puneij(10, 12); k = i*j;}

    // aici poate avea acces la arataij()
    void aratatot() {cout << k << " "; arataij();}
};

main()
{
    derivat ob;

//  ob.puneij(2, 3); // ilegal, puneij() este membru
//                      protejat al lui derivat
```



```
    ob.punek(); // OK, membru public al lui derivat
    ob.aratatot(); // OK, membru public al lui derivat

//  ob.arataij(); // ilegal, arataij() este membru
//                   protejat al lui derivat

    return 0;
}
```

După cum puteţi vedea citind comentariile, chiar dacă **puneij()** şi **arata()** sunt membri publici în **baza**, ei devin membri protejaţi în **derivat** atunci când aceasta îi moşteneşte folosind specificatorul de acces **protected**. Deci, aceasta înseamnă că ele nu vor fi accesibile pentru **main()**.

Moştenirea din clase de bază multiple

Este posibil pentru o clasă derivată să moştenească două sau mai multe clase de bază. Iată, de exemplu, **derivat** moşteneşte atât **baza1**, cât şi **baza2**:

```
// Un exemplu de clase de baza multiple.
#include <iostream.h>

class baza1 {
protected:
    int x;
public:
    void aratax() {cout << x << "\n";}
};

class baza2 {
protected:
    int y;
public:
    void aratay() {cout << y << "\n";}
};

// Mostenire din multiple clase de baza.
class derivat: public baza1, public baza2 {
public:
    void pune(int i, int j) {x=i; y=j;}
};

main()
```

```
{
    derivat ob;

    ob.pune(10, 20); // asigurat prin derivat
    ob.aratax(); // din baza1
    ob.aratay(); // din baza2

    return 0;
}
```

După cum ilustrează exemplul, pentru a moşteni mai mult decât o clasă, folosiţi o listă separată prin virgulă şi aveţi grijă să utilizaţi un specificator de acces pentru fiecare bază moştenită.

Constructori, destructori şi moştenire

Există două mari întrebări care se ridică relativ la constructori şi destructori atunci când este vorba de moştenire. Prima, când sunt apelate funcţiile constructor şi destructor din clasele de bază şi cele derivate? A doua, cum pot fi transmişi parametrii către funcţiile constructor din clasele de bază? Acest paragraf examinează cele două caracteristici foarte importante.

Când sunt executate funcţiile constructor şi destructor

Este posibil ca o clasă de bază, una derivată sau ambele să conţină funcţii constructor şi/sau destructor. Este important să înţelegeţi ordinea în care sunt executate aceste funcţii atunci când este creat un obiect al clasei derivate sau când încetează să mai existe. Examinaţi, pentru început, acest scurt program:

```
#include <iostream.h>

class baza {
public:
    baza() {cout << "Construieste baza\n";}
    ~baza() {cout << "Distruge baza\n";
};

class derivat: public baza {
public:
    derivat() {cout << "Construieste derivat\n";
    ~derivat() {cout << "Distruge derivat\n";}
};
```



```
main()
{
    derivat ob;
    // nu face nimic, decit sa construiasca si sa distruga ob
    return 0;
}
```

După cum precizează comentariul din **main()**, acest program pur şi simplu construieşte şi distruge un obiect numit **ob** care este de clasa **derivat**. Când se execută, acest program afişează:

```
Construieste baza
Construieste derivat
Distruge derivat
Distruge baza
```

După cum puteţi vedea, primul este executat constructorul pentru **baza**, urmat de cel pentru **derivat**. Apoi (deoarece **ob** este distrus imediat în acest program), este apelat destructorul lui **derivat**, urmat de cel pentru **baza**.

Rezultatul experimentului anterior poate fi generalizat. Când este creat un obiect al unei clase derivate, dacă pentru clasa de bază există un constructor, el va fi apelat primul, urmat de constructorul clasei derivate. Când este distrus un obiect al clasei derivate, primul este apelat destructorul său, urmat de cel al clasei de bază, dacă există. Altfel spus, funcţiile constructor sunt executate în ordinea derivării, iar funcţiile destructor în ordine inversă derivării.

Dacă vă gândiţi, există un motiv pentru care funcţiile constructor sunt executate în ordinea derivării. Deoarece o clasă de bază nu ştie nimic despre nici o clasă derivată, este necesar ca orice iniţializare să se facă independent şi anterior oricărei cerinţe anterioare de iniţializare efectuate de clasa derivată. De aceea ea trebuie executată prima.

Tot aşa, este logic ca funcţiile destructor să fie executate în ordinea inversă derivării. Deoarece o clasă de bază conţine o clasă derivată, distrugerea obiectului de bază implică distrugerea obiectului derivat. De aceea, destructorul derivat trebuie să fie apelat înainte ca obiectul să fie distrus complet.

În cazurile de moştenire multiplă (atunci când o clasă derivată devine clasă de bază pentru altă clasă derivată), se aplică regula generală: constructorii sunt apelaţi în ordinea derivării, iar destructorii în ordine inversă. De exemplu, acest program

```
#include <iostream.h>

class baza {
public:
```

```
    baza() {cout << "Construieste baza\n";}
    ~baza() {cout << "Distruge baza\n";}
};

class derivat1 : public baza {
public:
    derivat1() {cout << "Construieste derivat1\n";
    ~derivat1() {cout << "Distruge derivat1\n";}
};

class derivat2 : public derivat1 {
public:
    derivat2() {cout << "Construieste derivat2\n";}
    ~derivat2() {cout << "Distruge derivat2\n";}
};

main()
{
    derivat2 ob;

    // construieste si distruge ob

    return 0;
}
```

afişează această ieşire:

```
Construieste baza
Construieste derivat1
Construieste derivat2
Distruge derivat2
Distruge derivat1
Distruge baza
```

Aceeaşi regulă generală se aplică în situaţii în care apar clase de bază multiple. De exemplu, acest program

```
#include <iostream.h>

class baza1 {
public:
    baza1() {cout << "Construieste baza1\n";}
    ~baza1() {cout << "Distruge baza1\n";}
```



```
};

class baza2 {
public:
    baza2() {cout << "Construieste baza2\n";}
    ~baza2() {cout << "Distruge baza2\n";}
};
class derivat : public baza1, public baza2 {
public:
    derivat() {cout << "Construieste derivat\n";}
    ~derivat() {cout << "Distruge derivat\n";}
};

main()
{
    derivat ob;

    // construieste si distruge ob

    return 0;
}
```

afişează această ieşire:

```
Construieste baza1
Construieste baza2
Construieste derivat
Distruge derivat
Distruge baza2
Distruge baza1
```

După cum puteţi vedea, constructorii sunt apelaţi în ordinea derivării - de la stânga la dreapta - după cum s-a specificat în lista de moştenire a clasei **derivat**. Destructorii sunt apelaţi în ordine inversă - de la dreapta la stânga. Aceasta înseamnă că, având specificată **baza2** înaintea clasei **baza1** în lista pentru **derivat**, aşa cum se arată aici:

```
class derivat: public baza2, public baza1 {
```

ieşirea acestui program va arăta astfel:

```
Construieste baza2
Construieste baza1
```

```
Construieste derivat
Distruge derivat
Distruge baza1
Distruge baza2
```

Transmiterea parametrilor spre constructorii clasei de bază

Până acum nici unul dintre exemplele anterioare nu a inclus funcții constructor cu argumente. În cazurile în care doar constructorul clasei derivate necesită unul sau mai mulți parametri, folosiți pur și simplu sintaxa constructorului parametrizat standard (vedeți **Capitolul 12**). Totuși, cum transmiteți argumentele unei funcții constructor din clasa de bază? Răspunsul este să folosiți o formă extinsă a declarației constructorului clasei derivate, care transmite argumentele unuia sau mai multor constructori din clasa de bază. Iată forma generală a declarației extinse pentru constructorul din clasa de bază:

```
constructor-derivat(listă-argumente) : baza1(lista-argumente),
                                       baza2(lista-argumente),
                                       .
                                       .
                                       .
                                       bazaN(lista-argumente)
{
    // corpul constructorului derivat
}
```

Aici, *baza1* până la *bazaN* sunt numele claselor de bază moștenite de clasa derivată. Rețineți că declarația funcției constructor a clasei derivate este separată de clasele de bază prin două puncte iar clasele de bază multiple sunt separate unele de celelalte prin virgule. Să luăm acest program:

```
#include <iostream.h>

class baza {
protected:
    int i;
public:
    baza(int x) {i=x; cout << "Construieste baza\n";}
    ~baza() {cout << "Distruge baza\n";}
};

class derivat: public baza {
    int j;
public:
    // derivat foloseste x; y este pasat bazei.
    derivat(int x, int y) : baza(y)
          {j=x; cout << "Construieste derivat\n";}
    ~derivat() {cout << "Distruge derivat\n";}
    void arata() {cout << i << " " << j << "\n";}
};

main()
{
  derivat ob(3, 4);

  ob.arata(); // afiseaza 4 3

  return 0;
}
```

Constructorul lui **derivat** este declarat aici ca preluând doi parametri, **x** și **y**. Dar, **derivat()** folosește doar **x**; **y** este pasat către **baza()**. În general, constructorul clasei derivate trebuie să declare atât parametrul/parametrii care este/sunt cerut/ceruți de el, cât și pe cel/cei ai clasei de bază. După cum ilustrează exemplul, toți parametrii ceruți de clasa de bază îi sunt transmiși acesteia prin constructorul clasei de bază specificate după două puncte.

Iată un exemplu care folosește clasele de bază multiple:

```
#include <iostream.h>

class baza1 {
protected:
    int i;
public:
    baza1(int x) {i=x; cout << "Construieste baza1\n";}
    ~baza1() {cout << "Distruge baza1\n";}
};

class baza2 {
protected:
    int k;
public:
    baza2(int x) {k=x; cout << "Construieste baza2\n";}
    ~baza2() {cout << "Distruge baza2\n";}
};
```

```
class derivat: public baza1, public baza2 {
    int j;
public:
    derivat(int x, int y, int z) : baza1(y), baza2(z)
        {j=x; cout << "Construieste derivat\n";}
    ~derivat() {cout << "Distruge derivat\n";}
    void arata() {cout << i << " " << j << " " << k << "\n";}
};

main()
{
derivat ob(3, 4, 5);

ob.arata(); // afiseaza 4 3 5

return 0;
}
```

Este important să înţelegeţi că argumentele unui constructor al clasei de bază sunt transmise prin intermediul argumentelor constructorului clasei derivate. De aceea, chiar în cazul în care constructorul unei clase derivate nu foloseşte nici un argument, va fi necesar să declare unul sau mai multe dacă clasa de bază preia unul sau mai multe argumente. În această situaţie, argumentele transmise clasei derivate sunt pur şi simplu transferate bazei. De exemplu, în programul următor, constructorul clasei derivate nu preia nici un argument, dar **baza1()** şi **baza2()** preiau.

```
#include <iostream.h>

class baza1 {
protected:
    int i;
public:
    baza1(int x) {i=x; cout << "Construieste baza1\n";}
    ~baza1() {cout << "Distruge baza1\n";}
};

class baza2 {
protected:
    int k;
public:
    baza2(int x) {k=x; cout << "Constriieste baza2\n";}
```



```
    ~baza2() {cout << "Distruge baza2\n";}
};

class derivat: public baza1, public baza2 {
public:
    /* Constructorul derivat nu foloseste parametri,
       dar trebuie totusi declarat ca preluandu-i
       pentru a-i pasa claselor de baza.
    */
    derivat(int x, int y) : baza1(y), baza2(y)
        {cout << "Construieste derivat\n";}
    ~derivat() {cout << "Distruge derivat\n";}
    void arata() {cout << i << " " << k << "\n";}
};

main()
{
derivat ob(3, 4);

ob.arata(); // afiseaza 3 4

return 0;
}
```

O funcţie constructor a clasei derivate este liberă să folosească oricare dintre parametrii pe care declară că îi preia, chia dacă unul sau mai mulţi sunt transmişi către o clasă de bază. Altfel spus, un argument care este transmis unei clase de bază nu exclude utilizarea sa şi de către o clasă derivată. De exemplu, acest fragment este perfect valid:

```
class derivat: public baza {
    int j;
public:
    // derivat foloseste atit x cit si y si apoi le paseaza
    // bazei.
    derivat(int x, int y): baza(x, y)
        {j = x*y; cout << "Construieste derivat\n";}
```

Încă un lucru de ţinut minte când transmiteţi argumente către constructorii din clasa de bază: argumentele pot să fie orice expresie validă în acel moment, inclusiv apelări de funcţii şi de variabile. Acest lucru este un aspect al faptului că C++ permite iniţializare dinamică.

Permiterea accesului

Când o clasă de bază este moştenită ca **private**, toţi membrii publici şi protejaţi ai acelei clase devin membri privaţi ai clasei derivate. Totuşi, în anumite circumstanţe, va trebui să refaceţi specificatorul de acces original al unuia sau mai multor membri moşteniţi. De exemplu, veţi dori să oferiţi statut public în cadrul clasei derivate anumitor membri publici ai clasei de bază, chiar dacă ea a fost moştenită ca **private**. Pentru a face acest lucru trebuie să folosiţi o *declarare de acces* în interiorul clasei derivate. O declarare de acces are forma generală:

clasa-de-baza::membru;

Declararea de acces este pusă în antetul de acces corespunzător din declararea clasei derivate. Reţineţi că într-o declarare de acces nu este necesară (şi nici permisă) o declarare de tip.

Pentru a vedea cum lucrează o declarare de acces, să începem cu acest scurt fragment:

```
class baza {
public:
    int j; // public in baza
};

// Mosteneste baza ca particular.
class derivat: private baza {
public:
    // aici se afla declararea accesului
    baza::j; // face j din nou public
    .
    .
    .
};
```

După cum ştiţi, deoarece **baza** este moştenită ca **private** în **derivat**, variabila publică **j** este făcută implicit variabilă particulară în **derivat**. Dar, dacă

```
baza::j;
```

este inclusă ca declaraţie de acces în antetul **public** al lui **derivat**, **j** este refăcut cu statutul său **public**.

Pentru a reface dreptul de acces la membrii publici şi protejaţi, puteţi folosi o declarare de acces. Însă, nu puteţi folosi o asemenea declarare pentru a creşte sau scădea statutul de acces la un membru. De exemplu, un membru declarat ca particular într-o clasă de bază nu poate fi făcut public într-o clasă derivată. (Dacă C++ ar permite să se întâmple aceasta, ar distruge mecanismul de încapsulare!)

Următorul program ilustrează declaraţia de acces.

```
#include <iostream.h>

class baza {
    int i; // particular in baza
public:
    int j, k;
    void punei(int x) {i = x;}
    int dai() {return i;}
};

// Mosteneste baza ca particular.
class derivat: private baza {
public:
    /* Urmatoarele trei instructiuni incalca
       mostenirea bazei ca particular si refac
       accesul public la j, punei() si dai(). */
    baza::j; // face din nou j public - dar nu si pe k
    baza::punei; // face public pe punei()
    baza::dai; // face public pe dai()

//  baza::i; // ilegal, nu puteti modifica accesul original
    int a; // public
};

main()
{
    derivat ob;
//  ob.i = 10; // ilegal deoarece i este particular in derivat

    ob.j = 20; // legal deoarece j este facut public in derivat
//  ob.k = 30; // ilegal deoarece j este particular in derivat

    ob.a = 40 ; // legal deoarece a este public in derivat
    ob.punei(10);

    cout << ob.dai() << " " << ob.j << " " << ob.a;

    return 0;
}
```

Reţineţi cum foloseşte programul declaraţiile de acces pentru a reface statutul public pentru **j**, **punei()** şi **dai()**.

Declaraţiile de acces sunt acceptate în C++ pentru a se adapta acelor situaţii în care majoritatea claselor moştenite sunt destinate să fie particulare, dar câţiva membri trebuie să îşi păstreze statutul lor public sau protejat.



NOTĂ: Deşi propunerea de standard ANSI C++ admite încă declaraţiile de acces, el descurajează utilizarea lor. Aceasta înseamnă că, deşi acesta încă le mai acceptă, este posibil să nu o mai facă în următoarele sale versiuni. În schimb, standardul sugerează realizarea aceluiaşi efect aplicând cuvântul cheie ***using****. (Instrucţiunea* ***using*** *va fi discutată mai departe.) Totuşi, acum, când scriu această carte, declaraţiile de acces sunt încă larg utilizate şi nici un compilator folosit uzual nu acceptă cuvântul cheie* ***using****.*

Clase de bază virtuale

Când sunt moştenite clase de bază multiple, într-un program în C++ se poate introduce un element de ambiguitate. De exemplu, să luăm acest program incorect:

```
// Acest program contine o eroare si nu va fi compilat.
#include <iostream.h>

class baza {
public:
    int i;
};

// derivat1 mosteneste baza.
class derivat1 : public baza {
public:
    int j;
};

// derivat2 mosteneste baza.
class derivat2 : public baza {
public:
    int k;
};

/* derivat3 mosteneste atat derivat1 cat si derivat2.
   Aceasta inseamna ca in derivat3 exista doua
   copii ale bazei! */
class derivat3 : public derivat1, public derivat2 {
public:
    int sum;
};

main(void)
{
    derivat3 ob;

    ob.i = 10; // este ambiguu, care dintre i???
    ob.j = 20;
    ob.k = 30;

    // i este aici, de asemenea, ambiguu
    ob.sum = ob.i + ob.j + ob.k;

    // de asemenea ambiguu, care dintre i?
    cout << ob.i << " ";

    cout << ob.j << " " << ob.k << " ";
    cout << ob.sum;

    return 0;
}
```


După cum indică şi comentariile din program, atât **derivat1** cât şi **derivat2** moştenesc **baza**. Dar, **derivat3** moşteneşte atât **derivat1** cât şi **derivat2**. Aceasta înseamnă că într-un obiect de tipul **derivat3** sunt prezente două copii pentru **baza**. De aceea, pentru o expresie ca

```
ob.i = 20;
```

nu se poate stabili despre care **i** este vorba - cel din **derivat1** sau cel din **derivat2?** Deoarece există două copii ale clasei **baza** în obiectul **ob**, există două **ob.i**.! După cum puteţi vedea, instrucţiunea este inerent ambiguă.

Există două căi de remediere a acestui program. Prima este să îi aplicaţi lui **i** operatorul de specificare a domeniului şi să îl selectaţi manual. De exemplu, această versiune a programului va fi compilată şi va rula conform aşteptărilor:

```
// Acest program foloseste explicit operatorul de declarare
// a domeniului pentru a-l selecta pe i.
#include <iostream.h>
```

```
class baza {
public:
    int i;
};

// derivat1 mosteneste baza.
class derivat1 : public baza {
public:
    int j;
};

// derivat2 mosteneste baza.
class derivat2 : public baza {
public:
    int k;
};

/* derivat3 mosteneste atat derivat1 cat si derivat2.
   Aceasta inseamna ca in derivat3 exista doua
   copii ale bazei! */
class derivat3 : public derivat1, public derivat2 {
public:
    int sum;
};

main(void)
{
    derivat3 ob;
    ob.derivat1::i = 10; // s-a rezolvat, foloseste i din
    //                      derivat1
    ob.j = 20;
    ob.k = 30;

    // s-a rezolvat
    ob.sum = ob.derivat1::i + ob.j + ob.k;

    // si aici s-a rezolvat
    cout << ob.derivat1::i << " ";

    cout << ob.j << " " << ob.k << " ";
    cout << ob.sum;

    return 0;
}
```


După cum puteţi vedea, deoarece s-a aplicat ::, acest program a selectat versiunea **derivat1** din **baza**. Totuşi, această soluţie ridică o problemă mai amplă: ce se întâmplă dacă este cerută efectiv doar o copie pentru **baza**? Există vreo cale de a preveni includerea a două copii în **derivat3**? Răspunsul, după cum probabil aţi ghicit, este da. Soluţia se obţine folosind clasele de bază **virtuale**.

Când două sau mai multe obiecte sunt derivate dintr-o clasă de bază comună, puteţi preveni prezenţa într-unul dintre acestea a mai multor copii ale clasei de bază, declarând-o **virtual** în momentul în care ea este moştenită. Obţineţi această precedând numele clasei de bază cu cuvântul cheie **virtual**. Iată, de exemplu, o altă versiune a programului, exemplu în care **derivat3** conţine doar o singură copie din **baza**:

```
// Acest program foloseste clase de baza virtuale.
#include <iostream.h>

class baza {
public:
    int i;
};

// derivat1 mosteneste baza ca virtal.
class derivat1 : virtual public baza {
public:
    int j;
};

// derivat2 mosteneste baza ca virtual.
class derivat2 : virtual public baza {
public:
    int k;
};

/* derivat3 mosteneste atat derivat1 cat si derivat2.
   De aceasta data exista doar o singura copie a clasei de
   baza. */
class derivat3 : public derivat1, public derivat2 {
public:
    int sum;
};

main(void)
{
    derivat3 ob;
```

```
        ob.i = 10; // acum nu este ambiguu
        ob.j = 20;
        ob.k = 30;

        // nu este ambiguu
        ob.sum = ob.i + ob.j + ob.k;

        // nu este ambiguu
        cout << ob.i << " ";

        cout << ob.j << " " << ob.k << " ";
        cout << ob.sum;

        return 0;
}
```

După cum puteți vedea, cuvântul cheie **virtual** precede restul specificației de acces a clasei moștenite. Acum, deoarece atât **derivat1** cât și **derivat2** au moștenit **baza ca virtual**, orice moștenire multiplă care le implică va determina prezența unei singure copii. De aceea, în **derivat3** există doar o singură copie din **baza**, iar **ob.i = 10** este perfect validă și nu este ambiguă.

Încă ceva de ținut minte: chiar dacă atât **derivat1** cât și **derivat2** specifică **baza** ca fiind **virtual**, **baza** este încă prezentă în obiectele de acel tip. De exemplu, următoarea secvență este perfect validă:

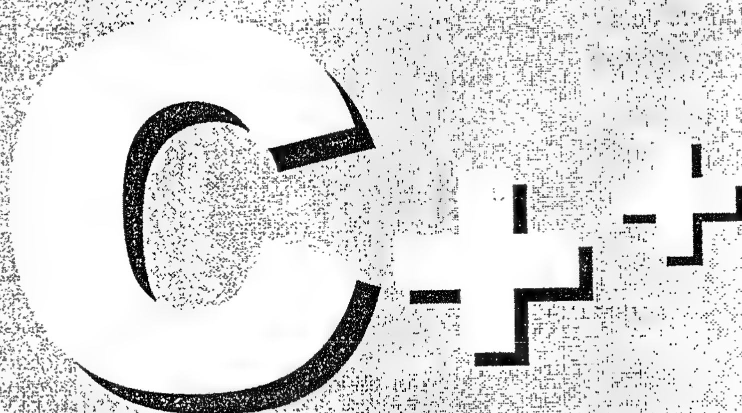
```
// defineste o clasa de tipul derivatl
derivatl clasamea;

clasamea.i = 88;
```

Singura diferență dintre o clasă normală și una virtuală este atunci când un obiect moștenește baza mai mult decât o singură dată. Dacă sunt folosite clase de bază virtuale, atunci o singură clasă de bază este prezentă în obiect. Altfel, vor exista copii multiple.

Capitolul 16

Funcții virtuale și polimorfism



Polimorfismul (o interfaţă, metode multiple) este admis de C++ atât în timpul compilării cât şi în timpul rulării. Polimorfismul din timpul compilării, obţinut prin funcţiile şi operatorii supraîncărcaţi, a fost discutat în **Capitolul 14**. Polimorfismul din timpul rulării este realizat folosind moştenirea şi funcţiile virtuale, acestea fiind, de altfel, subiectele acestui capitol.

Funcţiile virtuale

O *funcţie virtuală* este o funcţie care este declarată ca fiind **virtual** în clasa de bază şi redefinită de o clasă derivată. Pentru a declara o funcţie ca fiind virtuală, declararea sa este precedată de cuvântul cheie **virtual**. Redefinirea funcţiei în clasa derivată modifică şi are prioritate faţă de definiţia funcţiei din clasa de bază. În esenţă, o funcţie virtuală declarată în clasa de bază acţionează ca un substitut pentru păstrarea datelor care specifică o clasă generală de acţiuni şi declară forma interfeţei. Redefinirea unei funcţii virtuale într-o clasă derivată oferă operaţiile efective pe care le execută funcţia. Altfel spus, o funcţie virtuală defineşte o clasă generală de acţiuni. Redefinirea ei introduce o metodă specifică.

Când sunt utilizate „normal", funcţiile virtuale se comportă exact ca oricare altă funcţie membru al clasei. Însă, ceea ce le face importante şi capabile să admită polimorfismul este modul în care se comportă când sunt apelate printr-un pointer. După cum s-a discutat în **Capitolul 13**, un pointer al clasei de bază poate fi folosit pentru a indica spre orice clasă derivată din acea bază. Când un astfel de pointer indică spre un obiect derivat care conţine o funcţie virtuală, C++ determină care dintre versiunile funcţiei să fie apelată, în funcţie de *tipul obiectului spre care indică* acel pointer. Astfel, când sunt indicate obiecte diferite, sunt executate diferite versiuni ale funcţiei virtuale.

Înainte de a mai discuta teoretic, să examinăm acest scurt exemplu:

```
#include <iostream.h>

class baza {
public:
    virtual void vfunc() {
          cout << "Aceasta este vfunc() din baza.\n";
    }
};
class derivat1 : public baza {
public:
    void vfunc() {
          cout << "Aceasta este vfunc() din derivat1.\n";
    }
};
```



```
class derivat2 : public baza {
public:
    void vfunc() {
          cout << "Aceasta este vfunc() din derivat2.\n";
    }
};

main()
{
    baza *p, b;
    derivat1 d1;
    derivat2 d2;

  // indica spre baza
  p = &b;
  p->vfunc(); // acces la vfunc() din baza

  // indica spre derivat1
  p = &d1;
  p->vfunc(); // acces la vfunc() din derivat1

  // indica spre derivat2
  p = &d2;
  p->vfunc(); // acces la vfunc() din derivat2

  return 0;
}
```

Acest program afişează următoarele:

```
Aceasta este vfunc() din baza
Aceasta este vfunc() din derivat1
Aceasta este vfunc() din derivat2
```

După cum ilustrează programul, funcţia **vfunc()** este declarată în interiorul clasei **baza**. Observaţi cum cuvântul cheie **virtual** precede restul declarării funcţiei. Când **vfunc()** este redefinit în **derivat1** şi în **derivat2**, **virtual** nu mai este necesar. (Totuşi, nu este o eroare să îl includeţi atunci când redefiniţi o funcţie virtuală în interiorul unei clase derivate.)

În acest program **baza** este moştenită atât de **derivat1** cât şi de **derivat2**. În interiorul fiecărei definiţii de clasă, **vfunc()** este redefinită relativ la acea clasă. În interiorul funcţiei **main()**, sunt declarate patru variabile:

Nume	Tip
p	Pointer al clasei de baza
b	Obiect din bază
d1	Obiect din derivat1
d2	Obiect din derivat2

Apoi, lui **p** îi este atribuită adresa lui **b**, iar **vfunc()** este apelat prin **p**. Deoarece **p** indică spre un obiect de tipul **baza**, este executată acea versiune a lui **vfunc()**. Apoi, lui **p** i se dă adresa lui **d1** şi, din nou, **vfunc()** este apelată folosind **p**. De această dată, **p** indică spre un obiect de tipul **derivat1**. Aceasta face să fie executată **derivat1::vfunc()**. În sfârşit, lui **p** îi este atribuită adresa lui **d2** şi, din nou, este apelat **vfunc()** iar **p->vfunc()** determină executarea versiunii **vfunc()** din **derivat2**. Esenţialul aici este că versiunea de **vfunc()** ce se va executa este stabilită de tipul de obiect spre care indică **p**. Mai mult, această determinare are loc în timpul rulării, iar procesul stă la baza polimorfismului din timpul rulării.

Deşi puteţi apela o funcţie virtuală în modul „obişnuit", folosind un nume de obiect şi operatorul punct, polimorfismul din timpul rulării este permis doar dacă accesul se face printr-un pointer al clasei de bază. De exemplu, bazându-ne pe programul precedent, următoarea instrucţiune este sintactic validă:

```
d2.vfunc(); // apeleaza vfunc() din derivat2
```

Dacă apelaţi o funcţie virtuală în acest fel, nu faceţi o greşeală, însă nici nu profitaţi de avantajul naturii virtuale a funcţiei **vfunc()**.

La prima vedere, redefinirea unei funcţii virtuale într-o clasă derivată pare similară cu supraîncărcarea funcţiei (overload). Totuşi, nu este aşa, iar operandul *supraîncărcare* nu se aplică redefinirii funcţiilor virtuale deoarece există mai multe diferenţe. Probabil că cea mai importantă este aceea că prototipul pentru o funcţie virtuală redefinită trebuie să coincidă cu cel specificat în clasa de bază. Aceasta diferă de supraîncărcarea unei funcţii normale, pentru care tipurile returnate şi numărul şi tipul parametrilor pot să difere. (De fapt, când supraîncărcaţi o funcţie, ori numărul, ori tipul parametrilor chiar *trebuie* să difere! Prin aceste diferenţe C++ poate să selecteze versiunea corectă a unei funcţii supraîncărcate.) Însă, când este redefinită o funcţie virtuală, toate caracteristicile prototipului său trebuie să fie aceleaşi. Dacă modificaţi prototipul atunci când încercaţi să redefiniţi o funcţie virtuală, compilatorul de C++ o va considera o simplă funcţie supraîncărcată, iar natura sa virtuală se va pierde. O altă restricţie importantă este aceea că funcţiile virtuale nu trebuie să fie membri de tip **static** ai clasei din care fac parte. Mai ales, ele nu pot fi **friend**. În sfârşit, funcţiile constructor nu pot fi virtuale, în schimb cele destructor pot fi.

Datorită restricţiilor şi diferenţelor dintre funcţiile supraîncărcate şi cele virtuale, pentru descrierea redefinirii funcţiei virtuale într-o clasă derivată se foloseşte operandul *suprascriere*.



NOTĂ: O clasă care include o funcţie virtuală este numită o clasă polimorfică.

Atributul virtual este moştenit

Când o funcţie virtuală este moştenită, se moşteneşte şi natura sa virtuală. Aceasta înseamnă că, atunci când o clasă derivată care a moştenit o funcţie virtuală este, ea însăşi, folosită ca o clasă de bază pentru o altă clasă derivată, funcţia virtuală poate fi în continuare suprascrisă. Altfel spus, o funcţie rămâne virtuală indiferent de câte ori este moştenită. De exemplu, să considerăm o versiune a programului precedent:

```
#include <iostream.h>

class baza {
public:
   virtual void vfunc() {
         cout << "Aceasta este vfunc() din baza.\n";
   }

};
class derivat1 : public baza {
public:
   void vfunc() {
         cout << "Aceata este vfunc() din derivat1.\n";
   }
};

/* derivat2 mosteneste functia virtuala vfunc()
   de la derivat1. */
class derivat2 : public derivat1 {
public:
   // vfunc() este inca virtuala
   void vfunc() {
         cout << "Aceasta este vfunc() din derivat2.\n";
   }
};

main()
{
   baza *p, b;
   derivat1 d1;
```

```
    derivat2 d2;

  // indica spre baza
  p = &b;
  p->vfunc(); // acces la vfunc() din baza

  // indica spre derivat1
  p = &d1;
  p->vfunc(); // acces la vfunc() din derivat1

  // indica spre derivat2
  p = &d2;
  p->vfunc(); // acces la vfunc() din derivat2

  return 0;
  }
```

După cum este de aşteptat, programul anterior afişează această ieşire:

```
Aceasta este vfunc() din baza
Aceasta este vfunc() din derivat1
Aceasta este vfunc() din derivat2
```

Funcţiile virtuale sunt ierarhizate

După cum ştiţi, când o funcţie este declarată ca fiind **virtual** într-o clasă de bază, ea poate fi suprascrisă de o clasă derivată. Totuşi, funcţia nu trebuie neapărat să fie suprascrisă. Dacă o clasă derivată nu suprascrie funcţia virtuală, atunci, când un obiect din acea clasă derivată are acces la funcţie, este folosită funcţia definită de clasa de bază. Să luăm, de exemplu, acest program:

```
#include <iostream.h>

class baza {
public:
    virtual void vfunc() {
          cout << "Aceasta este vfunc() din baza.\n";
    }

};
class derivat1 : public baza {
public:
```



```
    void vfunc() {
          cout << "Aceata este vfunc() din derivat1.\n";
    }
};

class derivat2 : public baza {
public:
// vfunc() nu este suprascrisa de derivat2, este folosita
// definitia din baza
};

main()
{
    baza *p, b;
    derivat1 d1;
    derivat2 d2;

    // indica spre baza
    p = &b;
    p->vfunc(); // acces la vfunc() din baza

    // indica spre derivat1
    p = &d1;
    p->vfunc(); // acces la vfunc() din derivat1

    // indica spre derivat2
    p = &d2;
    p->vfunc(); // foloseste vfunc() din baza

    return 0;
}
```

Programul precedent produce această ieşire:

```
Aceasta este vfunc() din baza.
Aceasta este vfunc() din derivat1.
Aceasta este vfunc() din baza.
```

Deoarece **derivat2** nu suprascrie **vfunc()**, atunci când **vfunc()** este apelată pentru obiecte de tipul **derivat2**, este folosită funcţia definită de **baza**.

Programul precedent ilustrează un caz special al unei reguli mai generale. Deoarece în C++ moştenirea este ierarhizată, este normal ca funcţiile virtuale să fie, de asemenea, ierarhizate. Aceasta înseamnă că, atunci când o clasă derivată

nu suprascrie o funcţie virtuală, este folosită prima redefinire găsită în ordinea inversă a derivării. De exemplu, în următorul program, **derivat2** este derivat din **derivat1**, care este derivat din baza. **derivat2** nu suprascrie **vfunc()**. Cum pentru **derivat2** cea mai apropiată versiune de **vfunc()** este cea din **derivat1**, atunci când un obiect din **derivat2** apelează **vfunc()** este folosită **derivat1::vfunc()**.

```
#include <iostream.h>

class baza {
public:
    virtual void vfunc() {
          cout << "Aceasta este vfunc() din baza.\n";
    }
};

class derivat1 : public baza {
public:
    void vfunc() {
          cout << "Aceasta este vfunc() din derivat1.\n";
    }
};

class derivat2 : public derivat1 {
public:
/* vfunc() nu este suprascrisa de derivat2.
   In acest caz, deoarece derivat2 este derivat
   din derivat1, este folosita vfunc() din derivat1.
*/
};

main()
{
    baza *p, b;
    derivat1 d1;
    derivat2 d2;

// indica spre baza
p = &b;
p->vfunc(); // acces la vfunc() din baza

// indica spre derivat1
p = &d1;
p->vfunc(); // acces la vfunc() din derivat1

// indica spre derivat2
p = &d2;
p->vfunc(); // acces la vfunc() din derivat1

return 0;
}
```


Programul precedent afişează următoarele:

```
Aceasta este vfunc() din baza.
Aceasta este vfunc() din derivat1.
Aceasta este vfunc() din derivat1.
```

Funcţii virtuale pure

Aşa cum ilustrează programul din exemplul precedent, când o funcţie virtuală nu este redefinită de o clasă derivată, va fi folosită versiunea definită în clasa de bază. Însă, în multe situaţii poate să nu existe nici o definiţie semnificativă a unei funcţii virtuale în cadrul clasei de bază. De exemplu, o clasă de bază poate să nu fie capabilă să definească suficient un obiect pentru a permite să fie creată o funcţie virtuală în acea clasă. Mai mult, în unele situaţii veţi dori să vă asiguraţi că toate clasele derivate suprascriu o funcţie virtuală. Pentru a trata aceste două cazuri, C++ admite funcţii virtuale pure.

O *funcţie virtuală pură* este o funcţie virtuală care nu are definiţie în clasa de bază. Pentru a declara o astfel de funcţie, folosiţi forma generală:

virtual *tip nume-functie(lista-de-parametri)* = 0;

Când o funcţie virtuală este construită pură, orice clasă derivată trebuie să-i asigure o definiţie. În cazul în care clasa derivată nu suprascrie funcţia virtuală pură, va rezulta o eroare în timpul compilării.

Următorul program conţine un exemplu simplu de funcţie virtuală pură. Tipul de bază, **numar**, conţine un întreg numit **val**, funcţia **puneval()** şi funcţia virtuală pură **arata()**. Clasele derivate **tiphex**, **tipdec** şi **tipoct** moştenesc **numar** şi redefinesc **arata()**, astfel încât ea afişează valoarea lui **val** în fiecare bază de numeraţie (hexazecimală, zecimală şi respectiv octală).

```
#include <iostream.h>

class numar {
protected:
```

```
    int val;
public:
    void puneval(int i) {val = i;}

    // arata() este o functie virtuala pura
    virtual void arata() = 0;
};

class tiphex : public numar {
public:
    void arata() {
            cout << hex << val << "\n";
    }
};

class tipdec : public numar {
public:
    void arata() {
            cout << val << "\n";
    }
};

class tipoct : public numar {
public:
    void arata() {
            cout << oct << val << "\n";
    }
};

main()
{
    tipdec d;
    tiphex h;
    tipoct o;

d.puneval(20);
d.arata();              // afiseaza 20 - zecimal

h.puneval(20);
h.arata();              // afiseaza 14 - hexazecimal

o.puneval(20);
o.arata();              // afiseaza 24 - octal
```



```
return 0;
}
```

Chiar dacă acest exemplu este foarte simplu, el ilustrează cum o clasă de bază poate să nu fie capabilă să definească semnificativ o funcţie virtuală. În acest caz, **numar** asigură doar o interfaţă obişnuită pentru tipurile derivate utilizate. De altfel, în acest exemplu, nu există nici un motiv să definim **arata()** în cadrul clasei **numar** deoarece scopul clasei derivate este să afişeze un număr în diferite baze de numeraţie. Desigur, puteţi să creaţi întotdeauna un substitut arbitrar de funcţie virtuală. Însă construind **arata()** ca funcţie pură, ne asigurăm, de asemenea, că toate clasele derivate o vor redefini într-adevăr, pentru a corespunde propriilor lor cerinţe.

Reţineţi că, atunci când o funcţie virtuală este declarată ca fiind pură, toate clasele derivate vor trebui să o suprascrie. Dacă o clasă derivată va omite acest lucru, va rezulta o eroare în timpul compilării.

Clase abstracte

O clasă care conţine cel puţin o funcţie virtuală pură se numeşte *abstractă*. Deoarece o clasă abstractă conţine una sau mai multe funcţii pentru care nu există definiţii (funcţii virtuale pure), nu pot fi create obiecte cu ajutorul ei; o clasă abstractă constituie un tip incomplet care este folosit ca fundament pentru clasele derivate.

Chiar dacă nu puteţi crea obiecte pentru clase abstracte, puteţi crea pointeri şi referinţe spre astfel de clase. Aceasta permite claselor abstracte să admită polimorfismul în timpul rulării, care constă din selectarea funcţiei virtuale corecte de către pointerii clasei de bază.

Utilizarea funcţiilor virtuale

După cum s-a menţionat, unul dintre aspectele centrale ale programării orientate pe obiecte este principiul „o interfaţă, metode multiple". Aceasta înseamnă că poate fi definită o clasă generală de acţiuni pentru care interfaţa este aceeaşi, iar fiecare exemplar (instanţă) specific al clasei generale care defineşte operaţiile efective corespunde propriei sale situaţii specifice. În termenii de C++, o clasă de bază poate fi folosită pentru a defini natura unei interfeţe cu o clasă generală. Fiecare clasă derivată introduce apoi operaţiile specifice cerute de tipurile de date folosite de tipul derivat.

Una dintre cele mai puternice şi mai flexibile căi de introducere a abordării „o interfaţă, metode multiple" este folosirea funcţiilor virtuale, a claselor abstracte şi a polimorfismului din timpul rulării. Folosind aceste caracteristici, creaţi o ierarhizare care trece de la general la specific (de la bază la derivat). Urmând această

filosofie, definiţi toate caracteristicile şi interfeţele comune ale unei clase de bază. În cazurile în care doar clasele derivate pot introduce anumite acţiuni, veţi folosi o funcţie virtuală pentru a defini interfaţa care va fi folosită de clasa sau clasele derivate. În esenţă, în clasa de bază creaţi şi definiţi tot ce se referă la cazul general. Clasa derivată completează detaliile specifice.

În continuare este dat un exemplu simplu, care ilustrează valoarea conceptului „o interfaţă, metode multiple". Pentru a efectua o conversie de unităţi dintr-un sistem într-altul (de exemplu, litri în galoane) este creată o ierarhie de clase. Clasa de bază **convert** declară două variabile, **val1** şi **val2**, care păstrează valorile iniţială şi respectiv convertită. Ea mai defineşte funcţiile **dainit()** şi **daconv()** care returnează valoarea iniţială şi cea convertită. Aceste elemente din **convert** sunt fixe şi aplicabile tuturor claselor derivate care vor moşteni **convert**. Dar, funcţia care va efectua efectiv conversia, **calcul()**, este o funcţie virtuală pură care trebuie să fie definită de clasa derivată din **convert**. Natura specifică a funcţiei **calcul()** va fi determinată de tipul de conversie care are loc.

```
// Exemplu practic de functie virtuala.
#include <iostream.h>

class convert {
protected:
    double val1; // valoare initiala
    double val2; // valoare convertita

public:
    convert(double i) {
          val1 = i;
    }
    double daconv() {return val2;}
    double dainit() {return val1;}

    virtual void calcul() = 0;
};

// Litri in galoane.
class l_in_g : public convert {
public:
    l_in_g(double i) : convert (i) { }
    void calcul() {
          val2 = val1 / 3.7854;
    }
};
```



```
// Fahrenheit in Celsius
class f_in_c : public convert {
public:
    f_in_c(double i) : convert(i) { }
    void calcul() {
          val2 = (val1-32) / 1.8;
    }
};

main()
{
    convert *p; // pointer spre clasa de baza

    l_in_g lgob(4);
    f_in_c fcob(70);

    // pentru conversie foloseste mecanismul functiei virtuale
    p = &lgob;
    cout << p->dainit() << " litri reprezinta ";
    p->calcul();
    cout << p->daconv() << " galoane\n"; // l_in_g
    p = &fcob;
    cout << p->dainit() << " in Fahrenheit reprezinta ";
    p = compute();
    cout << p->daconv() << " Celsius\n"; // f_in_c
    return 0;
}
```

Acest program creează două clase derivate din **convert**, numite **l_in_g** şi **f_in_c**. Ele efectuează conversiile pentru litri în galoane şi respectiv pentru grade Fahrenheit în grade Celsius. După cum puteţi vedea, pentru a efectua conversia dorită, fiecare clasă derivată suprascrie **calcul()** în felul ei propriu. Totuşi, deşi conversia efectivă (deci, metoda) diferă între **l_in_g** şi **f_in_c**, interfaţa rămâne constantă.

Una dintre calităţile claselor derivate şi ale funcţiilor virtuale este aceea că tratarea cazurilor noi se realizează foarte uşor. De exemplu, având programul anterior, puteţi să adăugaţi o conversie de picioare în metri incluzând această clasă:

```
// Picioare in metri
class f_in_m : public convert {
public:
    f_in_m(double i) : convert(i) { }
```

```
    void calcul() {
          val2 = val1 / 3.28;
    }
};
```

O utilizare importantă a claselor abstracte şi a funcţiilor virtuale este cea a *bibliotecilor de clase*. Puteţi crea o bibliotecă de clase generice şi extensibile care vor fi folosite de alţi programatori. Programatorul va moşteni clasa generală creată de dvs., care defineşte interfaţa şi elementele comune, şi va defini doar acele funcţii specifice clasei derivate. Prin bibliotecile de clase sunteţi capabili să creaţi şi să controlaţi interfaţa unei clase generale, permiţând în continuare altor programatori să o adapteze situaţiilor lor specifice.

În încheiere: clasa de bază **convert** este un exemplu de tip incomplet. Funcţia virtuală **calcul()** nu este definită în interiorul clasei **convert** deoarece nu se poate asigura o definire semnificativă. Clasa **convert** pur şi simplu nu conţine suficiente informaţii pentru a se putea defini **calcul()**. Se creează un tip complet doar când **convert** este moştenit de o clasă derivată.

Legături iniţiale / ulterioare

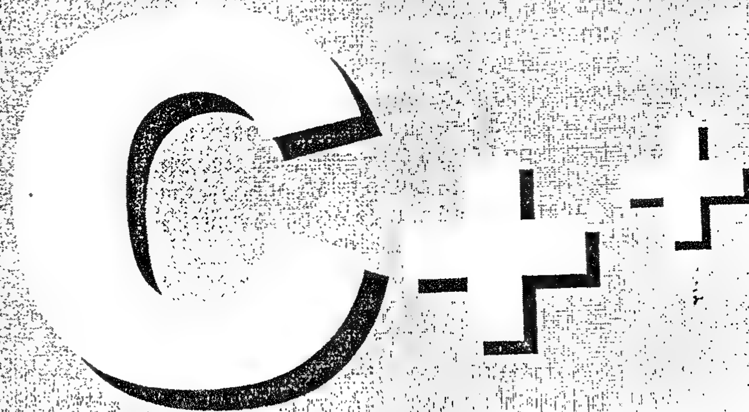
Înainte de a încheia acest capitol despre funcţii virtuale şi polimorfism în timpul rulării, mai există de definit doi operanzi frecvent utilizaţi în discuţiile despre C++ şi programarea orientată pe obiecte. Ei sunt *legături iniţiale* (early binding) şi *legături ulterioare* (late binding).

Legături iniţiale se referă la evenimentele care apar în timpul compilării. În esenţă, ele semnifică faptul că toate informaţiile necesare pentru a apela o funcţie sunt cunoscute în timpul compilării. (Altfel spus, legăturile iniţiale arată că legăturile dintre un obiect şi o apelare de funcţie se realizează în timpul compilării.) Exemplele de legături iniţiale includ apelările de funcţii normale (inclusiv funcţiile standard de bibliotecă), apelări de funcţii supraîncărcate şi operatori de supraîncărcare. Avantajul principal al legăturilor iniţiale este eficienţa. Deoarece toate informaţiile necesare pentru a apela o funcţie sunt determinate în timpul compilării, aceste tipuri de funcţii sunt foarte rapide.

Opusul legăturilor iniţiale sunt *legăturile ulterioare*. În C++, legăturile ulterioare se referă la apelările de funcţii care nu sunt rezolvate până în momentul rulării. Pentru realizarea legăturilor ulterioare sunt folosite funcţiile virtuale. După cum ştiţi, când accesul se face printr-un pointer din bază, apelarea efectivă a funcţiei virtuale este determinată de tipul obiectului spre care indică pointerul. Deoarece în majoritatea cazurilor acesta nu poate fi determinat în timpul compilării, obiectul şi funcţia nu sunt legate până în momentul rulării. Avantajul principal al legăturilor ulterioare este flexibilitatea. Spre deosebire de legăturile iniţiale, cele ulterioare vă permit să creaţi programe care să răspundă la evenimentele ce apar în timpul rulării fără să trebuiască să creaţi o cantitate mare de „cod pentru contingenţe". Reţineţi că, deoarece o apelare de funcţie nu este rezolvată până în momentul rulării, legătura ulterioară va determina timpi de execuţie ceva mai lenţi.

Capitolul 17

Bazele sistemului de I/O din C++



Pe lângă faptul că permite sistemul de I/O din C, C++ defineşte sistemul său propriu orientat pe obiecte. Ca şi sistemul de I/O din C, cel din C++ este complet integrat. Aceasta înseamnă că diferitele aspecte ale sistemului de I/O din C++, cum ar fi I/O de la consolă sau I/O de pe disc, sunt doar perspective diferite ale aceluiaşi mecanism. Acest capitol discută bazele sistemului de I/O orientat pe obiecte din C++. Chiar dacă exemplele din acest capitol folosesc I/O de la „consolă", informaţiile sunt aplicabile şi altor echipamente, inclusiv fişierelor de pe disc (discutate în **Capitolul 18**).

După cum ştiţi, sistemul de I/O din C este foarte bogat, flexibil şi puternic. Vă puteţi întreba de ce C++ introduce un alt sistem. Răspunsul este acela că sistemul de I/O din C nu cunoaşte obiectele. De aceea, pentru ca C++ să asigure un suport complet pentru programarea orientată pe obiecte, a fost necesar să se creeze un sistem de I/O orientat pe obiecte, care să poată opera cu obiectele create de utilizator. În afară de admiterea obiectelor, mai există şi alte câteva efecte secundare benefice ale utilizării sistemului de I/O din C++, chiar şi în programele care nu folosesc extensiv (sau chiar de loc) obiecte definite de utilizator. Veţi vedea câteva exemple mai departe, în acest capitol.

În acest capitol, veţi învăţa despre modul de formatare a datelor. Veţi mai învăţa despre cum se suprapun operatorii din C++ << şi >> astfel încât să poată fi folosiţi împreună cu clasele pe care le creaţi. De asemenea, veţi vedea cum se creează funcţiile speciale de I/O, numite manipulatori, care pot face programul dvs. mai eficient.

Streamuri în C++

Ca şi sistemul de I/O din C, cel din C++ operează prin streamuri. Ele au fost prezentate în detaliu în **Capitolul 9**; discuţia nu se va mai relua aici. Totuşi, să rezumăm: un stream este o entitate logică ce produce sau primeşte informaţie. Un stream este legat de un echipament fizic prin sistemul de I/O din C++. Toate streamurile se comportă în acelaşi fel, chiar dacă echipamentele fizice la care sunt conectate efectiv pot să difere substanţial. Deoarece toate streamurile se comportă la fel, aceleaşi funcţii de I/O din C++ pot opera, teoretic, asupra oricărui tip de echipament fizic. De exemplu, puteţi folosi aceeaşi funcţie care scrie într-un fişier şi pentru a scrie la imprimantă sau pe ecran. Avantajul acestei facilităţi este că trebuie să învăţaţi doar o singură interfaţă.

Clasele de bază pentru streamuri

C++ asigură suportul pentru sistemul său de I/O în fişierul antet IOSTREAM.H.Aici sunt definite două ierarhii de clase care admit operaţii de I/O. Clasa cu nivelul cel mai mic se numeşte **streambuf** şi asigură operaţiile de bază de intrare şi de ieşire. Nu veţi folosi **streambuf** direct, decât dacă veţi deriva propriile clase de I/O. A doua ierarhie porneşte cu clasa **ios**, care acceptă I/O formatate. Din ea sunt derivate clasele **istream**, **ostream** şi **iostream**. Aceste clase sunt folosite pentru a crea streamuri capabile să introducă, să obţină şi respectiv să introducă/să obţină. După cum veţi vedea în capitolele următoare, din **ios** sunt derivate multe alte clase pentru utilizarea fişierelor de pe disc şi formatarea în RAM.

Clasa **ios** conţine multe funcţii membre şi variabile membre care controlează sau urmăresc operaţiile fundamentale ale streamului. În cursul acestui capitol şi al următorului, se vor face multe referiri la membrii săi. Reţineţi doar că dacă folosiţi sistemul de I/O din C++ în manieră normală, membrii clasei **ios** vor fi capabili să lucreze cu orice stream.

Streamuri predefinite în C++

Când îşi începe execuţia un program în C++, se deschid automat patru streamuri încorporate. Ele sunt

Stream	Semnificaţie	Echipament implicit
cin	Intrare standard	Tastatură
cout	Ieşire standard	Ecran
cerr	Ieşire standard pentru eroare	Ecran
clog	Versiune cu memorie tampon pentru cerr	Ecran

Streamurile **cin**, **cout** şi **cerr** corespund streamurilor **stdin**, **stdout** şi **stderr** din C.

Implicit, streamurile standard sunt folosite pentru a comunica cu consola. Însă, în mediile care admit redirecţionarea I/O (cum ar fi DOS, Unix, OS/2 şi Windows), streamurile standard pot fi redirecţionate spre alte echipamente sau fişiere. Totuşi, pentru simplicitate, exemplele din acest capitol presupun că nu apare nici o redirecţionare I/O.

*NOTĂ: Standardul propus ANSI C++ mai defineşte următoarele patru streamuri: **win**, **wout**, **werr** şi **wlog**. Ele sunt versiunile streamurilor standard pentru caractere mari (wide caracters). Acestea sunt de tipul **wchar_t** şi, în general, au mărimea de 16 biţi; ele sunt folosite pentru seturile lărgite de caractere necesare anumitor limbi.*

I/O formatate

Sistemul de I/O din C++ vă permite să formataţi operaţiile de I/O. De exemplu, puteţi să daţi mărimea unui câmp, să specificaţi baza unui număr sau să determinaţi câte cifre se vor afişa după punctul zecimal. În esenţă, orice format pe care puteţi să îl obţineţi sau să-l introduceţi cu funcţiile din C **printf()** şi **scanf()** poate fi, de asemenea, obţinut sau introdus folosind operatorii de I/O din C++, << şi >>.

Există două căi înrudite, dar conceptual diferite, prin care puteţi să formataţi datele. În primul rând, puteţi avea acces direct la diferiţi membri ai clasei **ios**. Mai concret, puteţi să controlaţi diverşi indicatori pentru format, definiţi în cadrul clasei **ios**, sau să apelaţi diferite funcţii membre din **ios**. În al doilea rând, puteţi folosi funcţii speciale numite *manipulatori*, care pot fi incluse în expresii de I/O.

Vom începe discuţia despre I/O formatate folosind funcţiile membre, **ios** şi indicatorii.

Formatarea folosind membrii ios

Fiecărui stream îi este asociat un set de indicatori pentru format care controlează unele dintre căile prin care sunt formatate informaţiile de către un stream. În **ios**, în mod normal, indicatorilor li se atribuie nume şi valoare prin enumerare, aşa cum se arată în următorul exemplu:

```
// indicatori de formatare din ios
enum {
    skipws = 0x0001,
    left = 0x0002,
    right = 0x0004,
    internal = 0x0008,
    dec = 0x0010,
    oct = 0x0020,
    hex = 0x0040,
    showbasa = 0x0080,
    showpoint = 0x0100,
    uppercase = 0x0200,
    showpos = 0x0400,
    scientific = 0x800,
    fixed = 0x1000,
    unitbuf = 0x2000,
};
```

Indicatorii de format asociaţi cu un stream sunt codificaţi sub o anumită formă de întregi lungi. Standardul propus pentru ANSI C++ specifică tipul indicatorilor de format ca fiind **fmtflags**. Dar, nici un compilator larg răspândit nu defineşte curent acest tip. (Desigur, toate o vor face în viitorul apropiat.) Din punct de vedere practic, este aproape sigur că **fmtflags** va fi un simplu nume pentru **typedef** dat cu un întreg lung. Această carte va folosi tipul **long** când se va referi la indicatorii de formatare deoarece acesta este tipul folosit curent de toate compilatoarele de C++ uzuale. Va trebui însă să verificaţi manualul compilatorului.

Când este activat indicatorul **skipws**, caracterele de spaţii libere din faţă (spaţii simple, de tabulare şi de linie nouă) sunt eliminate atunci când se efectuează intrarea unui stream. Când **skipws** este şters, caracterele libere nu sunt eliminate.

Când este activat indicatorul **left**, ieşirea este aliniată la stânga. Când este activat **right**, ieşirea este aliniată la dreapta. Când **internal** este activ, o valoare numerică dintr-un câmp este completată cu spaţii inserate pornind de la semn sau de la caracterul bazei. (Veţi învăţa, în curând, cum să specificati mărimea unui câmp.) Dacă nici unul dintre acestea nu este activ, ieşirea este aliniată implicit la dreapta.

Tot implicit valorile numerice sunt obţinute în sistemul zecimal. Dar, puteţi modifica baza numărului. Activarea indicatorului **oct** determină ca ieşirea să fie afişată în octal. Activarea indicatorului **hex** face ca rezultatul să fie afişat în hexazecimal. Pentru a reveni la ieşirea în sistem zecimal, activaţi indicatorul **dec**. Aceste indicatoare determină şi baza când sunt introduse valori întregi.

Activarea indicatorului **showbase** determină afişarea bazei numărului. De exemplu, dacă baza de conversie este în hexazecimal, valoarea F1 va fi afişată ca 0x1F.

Când este afişată notaţia ştiinţifică, „e" este implicit scris cu literă mică. De asemenea, când se afişează o valoare hexazecimală, „x" este cu literă mică. Când este activ **uppercase**, aceste caractere sunt afişate cu literă mare.

Activarea indicatorului **showpos** determină ca în faţa valorilor pozitive să fie afişat un semn plus.

Activarea indicatorului **showpoint** determină, pentru ieşirile în virgulă mobilă, afişarea punctului zecimal şi a zerourilor finale - indiferent dacă sunt semnificative sau nu.

Activarea indicatorului **scientific** determină ca valorile în virgulă mobilă să fie afişate în notaţie ştiinţifică. Când este activat **fixed**, numerele în virgulă mobilă apar în notaţia normală, implicit, cu şase locuri pentru zecimale. Dacă nu este activ nici un indicator, compilatorul alege o metodă corespunzătoare.

Când este activ **unitbuf**, sistemul de I/O din C++ este golit după fiecare operaţie de ieşire.

NOTĂ: *Indicatorii de formatare descrişi mai înainte vor fi admişi de orice compilator de C++. Dar, în momentul scrierii acestei cărţi, natura exactă a indicatorilor de formatare din **ios** este încă în curs de definire de către comitetul de standardizare ANSI C++. De exemplu, a fost adăugat numele **boolalpha**, care permite operaţii de I/O asupra noului tip de date definit, **bool**. Acest indicator nu este definit curent de majoritatea compilatoarelor. Verificaţi manualul compilatorului dvs. pentru a vedea dacă sunt accesibile pentru utilizare acesta sau alţi indicatori de format.*

Activarea indicatorilor de format

Pentru a activa un indicator de format, folosiţi funcţia **setf()**. Această funcţie este membru al clasei **ios**. Iată forma sa cea mai uzuală:

long setf(long *indicator*);

Funcția returnează valorile precedente ale indicatorului de format şi activează acei indicatori specificaţi ca argumente; ceilalţi rămân neschimbaţi.
De exemplu, pentru a activa **showpos** puteţi folosi această instrucţiune:

```
stream.setf(ios::showpos);
```

Aici, *stream* este streamul pe care doriţi să îl modificaţi. De exemplu, următorul program afişează valoarea lui 100 în hexazecimal şi îi indică baza.

```
#include <iostream.h>

main()
{
    cout.setf(ios::hex);
    cout.setf(ios::showbase);

    cout << 100; // afiseaza 0x64

    return 0;
}
```

Este important să înţelegeţi că **setf()** este o funcţie membră a clasei **ios** şi are efect asupra streamurilor create de acea clasă. De aceea, orice apelare pentru **setf()** este făcută relativ la un anumit stream. Se poate invoca **setf()** în abstract. Altfel spus, în C++ nu există ideea de stare de format global. Fiecare stream îşi întreţine propriile informaţii referitoare la starea formatului.
Chiar dacă nu este practic greşit, există o cale mai eficientă de a scrie programul anterior. În loc să apelăm **setf()** de mai multe ori, puteţi să uniţi prin OR (SAU logic) valorile indicatorilor pe care doriţi să-i activaţi. De exemplu, următoarea instrucţiune realizează singură acelaşi lucru.

```
// Puteti uni cu OR doi sau mai multi indicatori
cout.setf(ios::showbasa | ios::hex);
```



*REŢINEŢI: Deoarece indicatorii de format sunt definiţi în interiorul clasei **ios**, pentru a avea acces la valorile lor trebuie să folosiţi **ios** şi operatorul de specificare a domeniului. De exemplu, **showbase** nu va fi recunoscută ca atare. Trebuie să specificaţi **ios::showbase**.*

Dezactivarea indicatorilor de format

Complementul funcţiei **setf()** este **unsetf()**. Această funcţie membru al lui **ios** este folosită pentru a şterge unul sau mai mulţi indicatori de format. Forma sa generală este:

long unsetf(long *indicatori*);

Sunt dezactivaţi indicatorii specificaţi în paranteze. (Toţi ceilalţi rămân neafectaţi.) Se returnează starea anterioară a indicatorilor.
Următorul program ilustrează **unsetf()**. Pentru început, el activează indicatorii **uppercase şi scientific**. Apoi el transpune 100.12 în notaţie ştiinţifică. În acest caz, E folosit la notaţia ştiinţifică este scris mare. După care dezactivează indicatorul **uppercase** şi scrie din nou 100.12 în notaţie ştiinţifică, folosind litera „e".

```
#include <iostream.h>

main()
{
    cout.setf(ios::uppercase | ios::scientific);

    cout << 100.12; // afiseaza 1.0012E+02

    cout.unsetf(ios::uppercase); // dezactiveaza uppercase

    cout << " \n" << 100.12; // afiseaza 1.0012e+02

    return 0;
}
```

O formă suprapusă a funcţiei setf()

Există o formă suprapusă a funcţiei **setf()** care are sintaxa generală:

long setf(long *indicatori1*, long *indicatori2*);

În această versiune, sunt afectaţi doar indicatorii specificaţi de *indicatori2*.. Ei sunt, pentru început, dezactivaţi şi apoi activaţi în funcţie de *indicatori1*. Reţineţi că, şi dacă *indicatori1* conţin şi alţi indicatori care nu sunt specificaţi de *indicatori2*, vor fi modificaţi doar cei ce apar în *indicatori2*. Este returnată starea anterioară a indicatorului. De exemplu, următorul program activează **showpos** şi **showpoint**. Apoi el îi dezactivează pe amândoi şi reactivează **showpos**.

```
#include <iostream.h>

main()
{

    cout.setf(ios::showpos | ios::showpoint);

    cout << 10.00 << " \n"; // afiseaza +10.000000

    cout.setf(ios::showpoint, ios::showpos |
                ios::showpoint);

    cout << 10.00;
    // showpos este dezactivat, se afiseaza 10.00000

    return 0;
}
```

Reţineţi că doar indicatorii specificaţi în *indicatori2* pot fi afectaţi de indicatorii specificaţi în *indicatori1*. De exemplu, următorul program nu va lucra.

```
// Acest program nu va lucra.
#include <iostream.h>

main()
{
    cout.setf(ios::showbase | ios::hex);

    cout << 100; // afiseaza 0x64

    cout.setf(ios::oct, ios::hex);
    // eroare, oct nu apare dupa virgula

    cout << " \n" << 100; // afiseaza 100 - nu 0144

    return 0;
}
```

Aici, în apelarea funcţiei **setf()**, *indicatori2* specifică faptul că doar **hex** poate fi afectat. Deoarece valoarea din *indicatori1* este **oct**, **hex** este dezactivat, dar **oct** nu este activat. Următorul program arată o versiune corectă.


```
// Acest program este acum corect.
#include <iostream.h>

main()
{
    // puteti sa alaturati prin SAU logic doi sau mai multi
    // indicatori
    cout.setf(ios::showbase | ios::hex);

    cout << 100; // afiseaza 0x64

    // acum oct poate fi activat
    cout.setf(ios::oct, ios::hex | ios::oct);

    cout << " \n" << 100; // afiseaza 0144

    return 0;
}
```

Se pot face referiri la oricare dintre câmpurile **oct**, **dec** şi **hex** prin numele colectiv **ios::basefield**. Ne putem referi similar şi la câmpurile **left**, **right** şi **internal** cu **ios::adjustfield**. În sfârşit, ne putem referi la câmpurile **scientific** şi **fixed** cu **ios::floatfield**. De exemplu, programul precedent ar putea fi scris astfel:

```
#include <iostream.h>

main()
{
    // puteti sa alaturati prin SAU logic doi sau mai multi
    // indicatori
    cout.setf(ios::showbasa | ios::hex);

    cout << 100; // afiseaza 0x64
    // foloseste ios::basefield
    cout.setf(ios::oct, ios::basefield);

    cout << " \n" << 100; // afiseaza 0144

    return 0;
}
```

Reţineţi că, de cele mai multe ori, veţi prefera să folosiţi **unsetf()** pentru a dezactiva indicatorii şi versiunea cu un singur parametru a funcţiei **setf()**, cea descrisă aici, pentru a-i activa. Versiunea **setf(long** *indicatori1*, **long** *indicatori2*)

este folosită în situaţii speciale. De exemplu, puteţi avea un şablon pentru un indicator care specifică starea tuturor indicatorilor de format, dar doriţi să modificaţi doar unul sau doi. În acest caz, puteţi specifica şablonul în *indicatori1* şi *indicatori2* pentru a indica pe care dintre aceştia îi veţi modifica.

Examinarea indicatorilor de format

Vor fi cazuri când veţi dori doar să cunoaşteţi starea curentă a indicatorilor de format dar să nu modificaţi nici unul. Pentru a realiza aceasta, **ios** include, de asemenea, funcţia membru **flags()**, care returnează pur şi simplu starea curentă a fiecărui indicator de format într-un număr întreg lung. Iată prototipul său:

long flags();

Următorul program foloseşte **flags()** pentru a afişa indicatorii de format relativi la **cout**. Fiţi atenţi, în special, la funcţia **showflags()**. Probabil că o veţi găsi folositoare pentru programele pe care le veţi scrie.

```
#include <iostream.h>

void showflags();

main()
{
    // arata starea implicita a indicatorilor de format
    showflags();

    cout.setf(ios::right | ios::showpoint | ios::fixed);

    showflags();

    return 0;
}

// Aceasta functie afiseaza starea indicatorilor de format.
void showflags()
{
    long f, i;
    int j;

    char indic[15][12] = {
          "skipws",
          "left",
          "right",
          "internal",
          "dec",
          "oct",
          "hex",
          "showbase",
          "showpoint",
          "uppercase",
          "showpos",
          "scientific",
          "fixed",
          "unitbuf",
    };

    f = cout.flags(); // obtine starea indicatorilor

    // verifica fiecare indicator
    for(i=1, j=0; i<=0x2000; i = i<<1, j++)
          if(i & f) cout << numgen[j] << " este activat \n";
          else cout << numgen[j] << " este dezactivat \n";

    cout << " \n";
}
```

Iată ieşirea acestui program:

```
skipws este activat
left este dezactivat
right este dezactivat
internal este dezactivat
dec este dezactivat
oct este dezactivat
hex este dezactivat
showbase este dezactivat
showpoint este dezactivat
uppercase este dezactivat
showpos este dezactivat
scientific este dezactivat
fixed este dezactivat
unitbuf este dezactivat

skipws este activat
left este dezactivat
```

```
right este activat
internal este dezactivat
dec este dezactivat
oct este dezactivat
hex este dezactivat
showbase este dezactivat
showpoint este activat
uppercase este dezactivat
showpos este dezactivat
scientific este dezactivat
fixed este activat
unitbuf este dezactivat
```

NOTĂ: De vreme ce este posibil ca un compilator de C++ să definească alte valori pentru indicatorii de format decât cele folosite în programul precedent, puteți să vedeți alte rezultate decât cele prezentate aici. Dacă este așa, verificați în manualul compilatorului dvs. ce valori folosește acesta.

Activarea tuturor indicatorilor

Funcția **flags()** are o a doua formă care vă permite să activați toți indicatorii de format asociați unui stream. Prototipul pentru această versiune a funcției **flags()** este prezentat aici:

long flags(long *f*);

Când folosiți această versiune, modelul de biți aflat în *f* este copiat în variabila folosită pentru a păstra indicatorii pentru format asociați acelui stream. De aceea, sunt afectați toți indicatorii. Funcția returnează starea anterioară.

Următorul program ilustrează această versiune a funcției **flags()**. Pentru început, ea construiește o grilă pentru indicatori care activează **showpos**, **showbase**, **oct** și **right**. Pentru majoritatea compilatoarelor de C++, ei au valorile 0x0400, 0x0080, 0x0020 și 0x0004. Când sunt grupați rezultă valoarea folosită în program, 0x04A4. Toți ceilalți indicatori sunt dezactivați. Apoi, se folosește **flags()** pentru a atribui aceste valori variabilei asociate cu **cout**. Funcția **showflags()** verifică dacă indicatorii au fost activați așa cum s-a indicat. (Este aceeași cu cea din programul anterior.)

```
#include <iostream.h>

void showflags();
```



```
main()
{
    // arata starea implicita a indicatorilor de format
    showflags();

    // showpos, showbase, oct, right sunt activate,
    // ceilalti nu
    long f = 0x04A4;
    cout.flags(f); // activeaza toti indicatorii

    showflags();

    return 0;
}
```

Utilizarea funcțiilor width(), precision() și fill()

În afară de indicatorii de format, în **ios** mai sunt definite trei funcții membre care activează următorii parametri pentru format: mărimea câmpului, precizia și caracterul de umplere. Funcțiile care execută aceste trei lucruri sunt **width()**, **precision()** și respectiv **fill()**. Ele vor fi examinate pe rând.

Implicit, când este obținută o valoare, ea ocupă doar atât spațiu câte caractere îi sunt necesare pentru a o afișa. Dar, puteți specifica o mărime de câmp minim folosind funcția **width()**. Prototipul ei este prezentat aici:

int width(int *w*);

w devine, aici, mărimea câmpului și este returnată mărimea anterioară a acestuia. Pentru unele implementări, mărimea câmpului trebuie specificată înainte de fiecare ieșire. Dacă nu este specificată, este folosită mărimea implicită a câmpului.

După ce ați specificat lungimea minimă a câmpului, când o valoare folosește mai puțin decât această lungime, pentru a se atinge lungimea specificată, câmpul va fi completat cu caracterul de umplere curent (implicit, spațiul). Rețineți însă că, dacă dimensiunea valorii este mai mare decât lungimea minimă a câmpului, câmpul va fi extins. Valorile nu vor fi trunchiate.

Când obțineți valori în virgulă mobilă, puteți determina numărul de cifre care să fie afișate după punctul zecimal folosind funcția **precision()**. Iată prototipul său:

int precision(int *p*);

Aici, precizia este stabilită prin *p* și se returnează valoarea anterioară. Precizia implicită este 6. În unele implementări precizia trebuie specificată înainte de

fiecare ieşire în virgulă mobilă. Dacă nu o specificaţi, este folosită precizia implicită.

Când un câmp trebuie să fie completat, el este umplut, implicit, cu spaţii. Dar, puteţi specifica un caracter de umplere folosind funcţia **fill()**. Prototipul său este:

char fill(char *ch*);

După o apelare a funcţiei **fill()**, *ch* devine noul caracter de umplere şi se returnează cel anterior.

Iată un program care ilustrează aceste funcţii:

```
#include <iostream.h>

main()
{
    cout.precision(4);

    cout.width(10);

    cout <, 10.12345 << "\n"; // afiseaza 10.12

    cout.fill('*');

    cout.width(10);
    cout << 10.12345 << "\n"; // afiseaza *****10.12

    // marimea cimpului se aplica, de asemenea, si sirurilor
    cout.width(10);
    cout << "Salut" << "\n"; // afiseaza *****Salut
    cout.width(10);
    cout.setf(ios::left); // aliniere la stanga
    cout << 10.12345; // afiseaza 10.12*****

    return 0;
    }
```

Aici este prezentată ieşirea acestui program:

```
     10.12
*****10.12
*****Salut
10.12*****
```


Utilizarea manipulatorilor pentru I/O formatate

A doua cale prin care puteţi modifica parametrii de formatare ai streamului este aceea de folosire a funcţiilor speciale numite *manipulatori*, care pot fi incluse în expresii de I/O. Manipulatorii standard sunt prezentaţi în **Tabelul 17-1**. După cum puteţi vedea examinând tabelul, mulţi manipulatori de I/O sunt paraleli funcţiilor din clasa **ios**.



NOTĂ: Manipulatorii prezentaţi în Tabelul 17-1 sunt admişi de toate compilatoarele de C++. Totuşi, în momentul scrierii acestei cărţi, natura exactă a manipulatorilor de I/O este în curs de definire de către comitetul de standardizare ANSI C++. De exemplu, manipulatorul ***boolalpha()*** *a fost adăugat pentru a permite operaţii de I/O pentru* ***bool****, noul tip de date definit. Acest manipulator s-ar putea să nu fie acceptat de compilatorul dvs.. Verificaţi în manualul compilatorului ce alţi manipulatori sunt disponibili.*

Pentru a avea acces la manipulatorii care preiau parametri (aşa cum este **setw()**), trebuie să includeţi în programul dvs. IOMANIP.H.

Iată un exemplu care foloseşte unii dintre manipulatori:

Manipulator	Scop	Intrare/Ieşire
dec	Intrare/ieşire de date în zecimal	Intrare şi ieşire
endl	Obţine un caracter de linie nouă şi goleşte streamul	Ieşire
ends	Obţine un null	Ieşire
flush	Goleşte un stream	Ieşire
hex	Introduce / obţine date în hexazecimal	Intrare şi ieşire
oct	Introduce / obţine date în octal	Intrare şi ieşire
resetiosflags(long *f*)	Dezactivează indicatorii specificaţi în f	Intrare şi ieşire
setbase(int *baza*)	Stabileşte baza ca *baza* numerică	Ieşire
setfill(int *ch*)	Stabileşte *ch* drept caracter de umplere	Ieşire
setiosflags(long *f*)	Activează indicatorii specificaţi în f	Intrare şi ieşire
setprecision(int *p*)	Stabileşte p ca număr de cifre pentru precizie	Ieşire
setw(int *w*)	Stabileşte *w* ca lungime a câmpului	Ieşire
ws	Omite spaţiile libere de la început	Intrare

Tabelul 17-1 *Manipulatorii în C++*

```
#include <iostream.h>
#include <iomanip.h>

main()
{
    cout << hex << 100 << endl;

    cout << setfill('?') << setw(10) << 2343.0;

    return 0;
}
```

Acesta afişează:

```
64
??????2343
```

Reţineţi cum apar manipulatorii în lanţul de operaţii de I/O. Reţineţi, de asemenea, că, atunci când un manipulator nu preia un argument, aşa cum este **endl()** din exemplu, el nu este urmat de paranteze (din cauză că operatorului supraîncărcat **<<** îi este transmisă adresa funcţiei).

Pentru comparaţie, iată o versiune echivalentă funcţional a programului precedent care foloseşte funcţiile membre din **ios** pentru a obţine aceleaşi rezultate:

```
#include <iostream.h>
#include <iomanip.h>

main()
{
    cout.setf(ios::hex);
    cout << 100 << "\n"; // 100 in hex

    cout.fill('?');
    cout.width(10);
    cout << 2343.0;

    return 0;
}
```

Aşa cum sugerează exemplul, avantajul principal al folosirii manipulatorilor în locul funcţiilor membre ale clasei **ios** este acela că ei vă permit deseori să scrieţi un cod mult mai compact.

Puteţi să folosiţi manipulatorul **setiosflags()** pentru a activa direct diverşii indicatori pentru format referitori la un stream. De exemplu, acest program foloseşte **setflags()** pentru a activa indicatorii **showbase şi showpos**:

```
#include <iostream.h>
#include <iomanip.h>
main()

{
    cout << setiosflags(ios::showpos);
    cout << setiosflags(ios::showbase);
    cout <, 123 << " " << hex << 123;

    return 0;
}
```

Manipulatorul **setiosflags()** efectuează aceeaşi acţiune ca şi funcţia membru **setf()**.

Supraîncărcarea operatorilor << şi >>

După cum ştiţi, operatorii << şi >> sunt supraîncărcaţi în C++ pentru a efectua operaţii de I/O pentru tipurile încorporate în acesta. Puteţi şi dvs. să supraîncărcaţi aceşti operatori astfel încât ei să efectueze operaţii de I/O asupra tipurilor pe care le creaţi.

În limbajul C++, operatorul de ieşire << este numit *operator de inserţie* deoarece el introduce caractere într-un stream. Similar, operatorul >> este numit *operator de extragere* deoarece el extrage caractere din acesta. Funcţiile operator care supraîncarcă inserţia şi extragerea sunt numite, în general, *de inserţie* şi, respectiv, *de extragere*.

Crearea propriilor dvs. funcţii de inserţie

Este foarte simplu să creaţi un operator de inserţie pentru o clasă pe care o construiţi. Toate funcţiile de inserţie au această formă generală:

```
ostream &operator<<(ostream &stream, tip_clasa obiect)
{
    // corpul funcţiei de inserţie
    return stream;
}
```

Reţineţi că funcţia returnează o referinţă spre un stream de tipul **ostream**. (Amintiţi-vă că **ostream** este o clasă derivată din **ios** care permite ieşirea.) Primul

parametru al funcţiei este o referinţă la streamul de ieşire. Al doilea parametru este obiectul care este inserat. (Al doilea parametru poate fi, şi el, o referinţă spre obiectul care este inserat.) Ultimul lucru pe care trebuie să-l facă funcţia înainte de a se încheia este să returneze un *stream*. Aceasta permite ca funcţia de inserţie să fie folosită într-un lanţ de inserţii.

În interiorul unei funcţii de inserţie puteţi să introduceţi orice fel de proceduri şi de operaţii pe care le doriţi. Aceasta înseamnă că este la latitudinea dvs. ce va face exact o asemenea funcţie. Totuşi, pentru ca ea să se încadreze într-o practică de programare corectă, ar trebui să limitaţi operaţiile efectuate de o funcţie de inserţie la introducerea de informaţii într-un stream. De exemplu, probabil că nu este o idee prea bună să aveţi o funcţie care să calculeze pi cu 30 de zecimale ca efect secundar al unei operaţii de inserţie.

Pentru a vedea un exemplu, să creăm o funcţie de inserţie pentru obiecte de tipul **agendatelefon**:

```
class agendatelefon {
public:
    char nume[80];
    int codzona;
    int prefix;
    int numar;
    agendatelefon(char *n, int a, int p, int nm)
    {
          strcpy(nume, n);
          codzona = a;
          prefix = p;
          numar = nm;
    }
};
```

Această clasă memorează numele persoanei şi numărul de telefon. Iată o cale de a crea o funcţie de inserţie pentru obiecte de tipul **agendatelefon**:

```
// Afiseaza numele si numarul de telefon
ostream &operator<<(ostream &stream, agendatelefon o)
{
    stream <, o.nume << " ";
    stream << "(" << o.codzona << ") ";
    stream << o.prefix << "-" << o.numar << "\n";

    return stream; // trebuie se returneze stream
}
```


Iată un scurt program care ilustrează funcţia de inserţie **agendatelefon**:

```
#include <iostream.h>
#include <stream.h>

class agendatelefon {
public:
    char nume[80];
    int codzona;
    int prefix;
    int numar;
    agendatelefon(char *n, int a, int p, int nm)
    {
          strcpy(nume, n);
          codzona = a;
          prefix = p;
          numar = nm;
    }
};

// Afiseaza numele si numarul de telefon.
ostream &operator<<(ostream &stream, agendatelefon o)
{
    stream << o.nume << " ";
    stream << "(" << o.codzona << ") ";
    stream << o.prefix << " -" << o.numar << "\n";

    return stream; // trebuie sa returneze stream
}

main()
{
    agendatelefon a("Ted", 111, 555, 1234);
    agendatelefon b("Alice", 312, 555, 5768);
    agendatelefon c("Tom", 212, 555, 9991);

    cout << a << b << c;

    return 0;
}
```

Programul are această ieşire:

```
Ted (111) 555-1234
Alice (312) 555-5768
Tom (212) 555-9991
```

Rețineți că, în programul precedent, funcția de inserție pentru **agendatelefon** nu este un membru al clasei **agendatelefon**. Chiar dacă acest lucru pare ciudat la prima vedere, motivul este ușor de înțeles. Când o funcție operator de orice fel este un membru al unei clase, operandul din stânga (transmis implicit prin **this**) este obiectul care generează apelarea acesteia. Mai mult, acest obiect este un *obiect din clasa* în care este membru funcția operator. Nu se poate schimba acest lucru. Dacă o funcție operator supraîncărcată este un membru al unei clase, termenul din stânga trebuie să fie un obiect din acea clasă. Totuși, când supraîncărcați inserțiile, operandul din stânga este un *stream* iar cel din dreapta este un obiect din acea clasă. De aceea, funcțiile de inserție supraîncărcate nu pot fi membri ai clasei pentru care sunt supraîncărcate. De asemenea, singurul motiv pentru care variabilele **nume**, **codzona**, **prefix** și **numar** sunt publice în programul precedent este ca ele să poată fi utilizate de o funcție de inserție care nu este membru al clasei lor.

Faptul că funcțiile de inserție nu pot fi membri ai clasei pentru care sunt definite pare a fi un handicap serios în C++. Dacă aceste funcții supraîncărcate nu sunt membre, cum pot să aibă acces la elementele particulare ale clasei? În programul anterior toți membrii au fost declarați publici. Totuși, încapsularea este o componentă esențială a programării orientate pe obiecte. Cerința ca toate datele care vor fi obținute folosind o funcție de inserție intră în conflict cu acest principiu. Din fericire, există o soluție a acestei dileme: declarați funcția de inserție un **friend** al clasei. Astfel se respectă cerința ca primul argument al funcției de inserție supraîncărcate să fie un stream și se garantează, în continuare, accesul ei la secțiunile particulare ale clasei pentru care este supraîncărcată. Iată același program modificat pentru a avea funcția de inserție ca funcție **friend**:

```
#include <iostream.h>
#include <stream.h>

class agendatelefon {
    // acum particulari
    char nume[80];
    int codzona;
    int prefix;
    int numar;
public:
    agendatelefon(char *n, int a, int p, int nm)
    {
          strcpy(nume, n);
```



```
          codzona = a;
          prefix = p;
          numar = nm;
    }
    friend ostream &operator<<(ostream &stream,
                               agendatelefon o);
};

// Afiseaza numele si numarul de telefon.
ostream &operator<<(ostream &stream, agendatelefon o)
{
    stream << o.nume << " ";
    stream << "(" << o.codzona << ") ";
    stream << o.prefix << "-" << o.numar << "\n";
    return stream; // trebuie sa returneze stream
}

main()
{
    agendatelefon a("Ted", 111, 555, 1234);
    agendatelefon b("Alice", 312, 555, 5768);
    agendatelefon c("Tom", 212, 555, 9991);

    cout << a << b << c;

    return 0;
}
```

Când definiți corpul unei funcții de inserție, amintiți-vă să îl faceți cât mai general posibil. Astfel, cel prezentat în exemplul anterior poate fi folosit cu orice stream deoarece își orientează ieșirea către **stream**, care este streamul care apelează acea funcție. Deși nu este, practic, greșit să scrieți linia

```
stream << o.nume < " ";
```

astfel:

```
cout < o.nume << " ";
```

ea va avea ca efect codificarea hard a streamului **cout** stream de ieșire. Versiunea inițială va lucra cu orice stream, inclusiv cu acelea legate de fișierele de pe disc. Deși, în unele situații, mai ales atunci când sunt implicate echipamente de ieșire speciale,va fi mai bine să codificați hardstreamul de ieșire, în majoritatea ocaziilor nu va fi nevoie. În general, cu cât vor fi mai flexibile funcțiile de inserție, cu atât vor fi mai valoroase.



NOTĂ: Funcţia de inserţie pentru clasa ***agendatelefon*** *lucrează bine cu excepţia cazurilor în care valoarea din* ***numar*** *este ceva de genul 0034, situaţie în care zerourile din faţă nu vor fi afişate. Pentru a rezolva aceasta, puteţi ori să convertiţi* ***numar*** *într-un şir, ori să stabiliţi caracterul de umplere ca fiind zero şi să folosiţi funcţia de format* ***width()*** *pentru a genera zerourile din faţă. Soluţia este lăsată ca exerciţiu, pentru cititor.*

Înainte de a trece la extractori, să urmărim încă un exemplu de funcţie de inserţie. O astfel de funcţie nu trebuie să fie limitată doar la manevrarea textelor. Ea poate fi folosită pentru a obţine date în orice formă care are sens. De exemplu, o funcţie de inserţie pentru o clasă care face parte din sistemul CAD poate emite instrucţiuni pentru plotter. Alta poate să genereze imagini grafice. O funcţie de inserţie pentru programele în Windows poate să afişeze casete de dialog. Pentru a „simţi" obţinerea altor obiecte, altele decât cele de tip text, să examinăm următorul program, care desenează casete pe ecran. (Deoarece nici C şi nici C++ nu definesc funcţii grafice, programul foloseşte caractere pentru a desena, dar sunteţi liberi să înlocuiţi cu elemente grafice, dacă sistemul vostru le admite.)

```
#include <iostream.h>
class caseta {
    int x, y;
public:
    caseta(int i, int j) {x=i; y=j;}
    friend ostream &operator<<(ostream &stream, caseta o);
};

// Afiseaza o caseta.
ostream &operator<<(ostream &stream, caseta o)
{
    register int i, j;

    for(i=0; i<o.x; i++)
          stream << "*";

    stream << "\n";

    for(j=1; j<o.y-1; j++) {
          for(i=0; i<o.x; i++)
                if(i==0 || i==o.x-1) stream << "*";
                else stream << " ";
          stream << "\n";
    }
    for(i=0; i<o.x; i++)
          stream << "*";
    stream << "\n";

    return stream;
}

main()
{

    caseta a(14, 6), b(30, 7), c(40, 5);
    cout << "Iata cateva casete:\n";
    cout << a << b << c <<;

    return 0;
}
```

Programul afişează următoarele:

```
Iata cateva casete:
**************
*            *
*            *
*            *
*            *
**************
******************************
*                            *
*                            *
*                            *
*                            *
*                            *
******************************
****************************************
*                                      *
*                                      *
*                                      *
****************************************
```

Crearea propriilor extractori

Extractorii sunt complemente ale funcţiilor de inserţie. Forma generală a funcţiilor de extracţie este:

```
istream &operator>>(istream &stream, tip_clasa &obiect)
{
    // corpul extractorului
    return stream;
}
```

Extractorii returnează o referință către un stream de tipul **istream**, care este un stream de intrare. Primul parametru trebuie să fie, de asemenea, o referință la un stream de tipul **istream**. Rețineți că al doilea parametru trebuie să se refere la un obiect de clasa pentru care este supraîncărcat extractorul. Aceasta permite ca obiectul să fie modificat de operația de intrare (extragere).

Continuând cu clasa **agendatelefon**, iată o cale de a scrie o funcție de extracție:

```
istream &operator>>(istream, agendatelefon &o)
{
    cout << "Introduceti nume: ";
    stream >> o.nume;
    cout << "Introduceti codzona: ";
    stream >> o.codzona;
    cout << "Introduceti prefix: ";
    stream >> o.prefix;
    cout << "Introduceti numar: ";
    stream >> o.numar;
    cout << "\n";

    return stream;
}
```

Rețineți că, deși aceasta este o funcție de intrare, ea efectuează și ieșire prin solicitarea către utilizator. Chiar dacă scopul principal al extractorului este intrarea, el poate efectua orice operație necesară pentru a-l atinge. Totuși, ca și pentru funcțiile de inserție, este mai bine să mențineți acțiunile efectuate de către extractor în sfera operațiilor de intrare. Dacă nu o veți face, riscați să pierdeți mult din structurare și din claritate.

Iată un program care ilustrează extractorul **agendatelefon**:

```
#include <iostream.h>
#include <string.h>

class agendatelefon {
    char nume[80];
```



```
    int codzona;
    int prefix;
    int numar;
public:
    agendatelefon() { };
    agendatelefon(char *n, int a, int p, int nm)
    {
        strcpy(nume, n);
        codzona = a;
        prefix = p;
        numar = nm;
    }
    friend ostream &operator<<(ostream &stream,
                               agendatelefon o);

    friend istream &operator>>(istream &stream,
                               agendatelefon &o);

};

// Afiseaza numele si numarul de telefon.
ostream &operator<<(ostream &stream, agendatelefon o)
{
    stream << o.nume << " ";
    stream << "(" << o.codzona << ") ";
    stream << o.prefix << "-" << o.numar << "\n";

    return stream; // trebuie sa returneze stream
}

// Introduce numele si numarul de telefon.
istream &operator >>(istream, agendatelefon &o)
{
    cout << "Introduceti nume: ";
    stream >> o.nume;
    cout << "Introduceti codzona: ";
    stream >> o.codzona;
    cout << "Introduceti prefix: ";
    stream >> o.prefix;
    cout << "Introduceti numar: ";
    stream >> o.numar;
    cout << "\n";

    return stream;
}
```

```
main()
{
    agendatelefon a;

    cin >> a;

    cout << a;

    return 0;
}
```

Crearea propriilor dvs. funcţii de manipulare

În afară de supraîncărcarea operatorilor de inserţie şi de extragere, puteţi să vă adaptaţi sistemul de I/O din C++ creând propriile dvs. funcţii de manipulare. Acestea sunt importante din două motive principale. Primul, puteţi să sintetizaţi o secvenţă de mai multe operaţii de I/O individuale într-o singură funcţie de manipulare. De exemplu, apar des situaţii în care într-un program se întâlneşte frecvent aceeaşi secvenţă de operaţii de I/O. În aceste cazuri, puteţi folosi un manipulator construit de dvs. pentru a efectua aceste acţiuni, simplificând astfel codul sursă şi prevenind erorile accidentale. Un manipulator propriu poate, de asemenea, să fie important când trebuie să efectuaţi operaţii de I/O pentru un echipament care nu este standardizat. De exemplu, puteţi să folosiţi un manipulator pentru a transmite codurile de control unui tip special de imprimantă sau unui sistem de recunoaştere optic.

Manipulatorii adaptaţi sunt caracteristici ale lui C++ care admit OOP, dar care pot totodată să trateze programe care nu sunt orientate pe obiecte. După cum veţi vedea, ei vă pot ajuta să faceţi orice program ce foloseşte mult operaţii de I/O mai clar şi mai eficient.

După cum ştiţi, există două tipuri de manipulatori: cei care operează cu streamuri de intrare şi cei care operează cu streamuri de ieşire. Dar, în afară de aceste două mari categorii, mai este şi o a doua împărţire: cei care preiau argumente şi cei care nu o fac. Există câteva diferenţe semnificative între modul în care sunt creaţi manipulatorii fără parametri şi cei parametrizaţi. Acest paragraf discută cum se creează ambele tipuri, începând cu manipulatorii fără parametri.

Crearea manipulatorilor fără parametri

Toate funcţiile de manipulare a ieşirilor fără parametri au această formă:

ostream &*nume-manip*(ostream &stream)
{
    // aici se afla codul dvs.
    return *stream*;
}

nume-manip este aici numele manipulatorului. Reţineţi că este returnată o referinţă la un stream de tip **ostream**. Acest lucru este necesar dacă manipulatorul este folosit ca parte a unei expresii mai mari de I/O. Este important să notaţi că, deşi manipulatorul are drept unic argument o referinţă la un stream asupra căruia operează, când manipulatorul este introdus într-o operaţie de ieşire nu este folosit nici un argument.

Ca un prim exemplu simplu, următorul program creează un manipulator numit **punehex()**, care activează indicatorul **showbase** şi obţine ieşirea în hexazecimal.

```
#include <iostream.h>
#include <iomanip.h>

// Un simplu manipulator de iesire.
ostream &punehex(ostream &stream)
{
    stream.setf(ios::showbase);

    stream.setf(ios::hex);
    return stream;
}

main()
{
    cout << 256 << " " << punehex << 256;
    return 0;
}
```

Acest program afişează **256 0x100**. După cum puteţi vedea, **punehex** este folosit ca parte a unei expresii de I/O în acelaşi fel ca şi oricare dintre manipulatorii încorporaţi.

Pentru a fi folositori, manipulatorii adaptaţi nu trebuie neapărat să fie complecşi. De exemplu, manipulatorii simpli **ss()** şi **sd()** afişează o săgeată la stânga şi respectiv una la dreapta, pentru a scoate ceva în evidenţă, aşa cum se prezintă aici:

```
#include <iostream.h>
#include <iomanip.h>
```

```
// Sageata la dreapta
ostream &sd(ostream &stream)
{
    stream << "-------> ";
    return stream;
}

// Sageata la stanga
ostream &ss(ostream &stream)
{
    stream << " <-------";
    return stream;
}

main()
{
    cout << "Bilant mare" << sd << 1233.33 << "\n";
    cout << "Fara acoperire " << sd << 567.66 << ss

    return 0;
}
```

Acest program afişează:

```
Bilant mare -------> 1233.23
Fara acoperire -------> 567.66 <-------
```

Dacă îi utilizaţi frecvent, aceşti manipulatori vă salvează de la tastări plicticoase.

Utilizarea unor funcţii de manipulare pentru ieşire este folositoare în special pentru a transmite coduri speciale către un echipament. De exemplu, o imprimantă poate fi capabilă să accepte diferite coduri care modifică mărimea tipului sau fontul, sau care plasează capul de scriere într-o anumită poziţie. Dacă aceste modificări trebuie făcute frecvent, atunci ele se pretează foarte bine prelucrării cu un manipulator.

Toate funcţiile de manipulare pentru intrare fără parametri au următoarea formă:

istream &*nume-manip*(istream &*stream*)
{
    // aici este codul dvs.
    return *stream*;
}

Un manipulator de intrare primeşte o referinţă spre streamul pentru care este apelat. Acest stream trebuie să fie returnat de către manipulator.

Următorul program creează manipulatorul de intrare **daparola()**, care sună clopoţelul şi apoi solicită o parolă.

```
#include <iostream.h>
#include <string.h>

// Un simplu manipulator de intrare.
istream &daparola(istream &stream)
{
    cout << '\a'; // suna clopotel
    cout << "Introduceti parola: ";

    return stream;
}

main()
{
    char par[80];

    do {
        cin >> daparola >> par;
    } while (strcmp(par, "parola"));

    cout << "Parola corecta\n";

    return 0;
}
```

Este esenţial ca manipulatorul să returneze un **stream**. Dacă nu o face, nu poate fi folosit într-o secvenţă de operaţii de intrare sau de ieşire.

Crearea manipulatorilor parametrizaţi

Crearea unei funcţii de manipulare care preia un argument este mai puţin intuitivă decât crearea uneia fără parametri. Un motiv pentru aceasta este că manipulatorii parametrizaţi folosesc clasele generice. Acestea sunt create folosindu-se cuvântul cheie **template**. (Şabloanele **(template)** şi clasele generice sunt discutate în **Capitolul 20**.) Dacă nu înţelegeţi cum operează clasele generice, nu veţi fi capabili să înţelegeţi pe deplin crearea manipulatorilor parametrizaţi. Un alt motiv pentru care manipulatorii parametrizaţi sunt mai complecşi este acela că metoda exactă pe care o folosiţi pentru a crea unul diferă de la compilator la compilator.

Deci, ce va funcţiona pentru un compilator nu va lucra, neapărat, şi la altul. Deşi comitetul ANSI C++ va finaliza acest subiect, nu toate compilatoarele introduc manipulatorii parametrizaţi în corespondenţă cu standardul. Datorită acestor probleme, dacă veţi crea proprii dvs. manipulatori parametrizaţi, va trebui să consultaţi manualul compilatorului pentru detalii specifice.



NOTĂ: Exemplele de manipulatori parametrizaţi prezentate în acest paragraf corespund propunerii de standard ANSI C++ şi vor funcţiona pentru majoritatea compilatoarelor de C++ moderne.

Pentru a crea un manipulator parametrizat trebuie să includeţi în fişierul dvs. IOMANIP.H. În acesta sunt definite trei clase generice: **omanip**, **imanip** şi **smanip**. **omanip** este folosit pentru a crea manipulatori de ieşire care preiau un argument. **imanip** este folosită pentru a crea manipulatori de intrare parametrizaţi ce pot fi folosiţi în streamuri de intrare şi de ieşire, iar **smanip** pentru manipulatori parametrizaţi ce acţionează atât asupra intrărilor cât şi a ieşirilor. (Pentru a vedea cum sunt implementate, probabil că va trebui să vedeţi definirea acestor clase din versiunea asigurată de compilatorul dvs. pentru IOMANIP.H.)

În general, întotdeauna când doriţi să obţineţi un manipulator care preia un argument, va trebui să creaţi două funcţii de manipulare supraîncărcate. În una dintre ele este necesar să definiţi doi parametri. Primul parametru este o referinţă către stream, iar al doilea este cel care va fi transmis funcţiei. A doua versiune a manipulatorului defineşte doar un singur parametru - cel care este specificat atunci când manipulatorul este folosit într-o expresie de I/O. Această a doua versiune generează o apelare a primei versiuni. Pentru manipulatorii de ieşire parametrizaţi veţi folosi aceste forme generale:

```
ostream &nume-manip(ostream &stream, tip param)
{
      // aici este codul dvs.
    return stream;
}
    // Supraîncărcare
    omanip<tip>nume-manip(tip param) {
          return omanip<tip>(nume-manip, param);
}
```

nume-manip este aici numele manipulatorului iar *tip* specifică tipul parametrului folosit de manipulator. Deoarece **omanip** este o clasă generică, *tip* devine, de asemenea, tipul de date asupra căruia operează obiectul **omanip** specific returnat de către manipulator.

Următorul program creează un manipulator cu parametri numit **compara()**, care indentează un rând cu numărul specificat de spaţii.


```
#include <iostream.h>
#include <iomanimp.h>

// Indenteaza cu numarul de spatii.
ostream &indent(ostream &stream, int lungime)
{
    register int i;

    for(i=0; i<lungime; i++) cout << " ";
    return stream;
}

omanip<int> indent(int lungime)
{
    return omanip<int>(indent, lungime);
}

main()
{
    cout << indent(10) << "Acesta este un test\n";
    cout << indent(20) << "pentru manipulatorul de
                            indentare.\n";
    cout << indent(5) << "Merge!\n";
    return 0;
}
```

După cum puteţi vedea, **indent()** este suprapusă aşa cum s-a descris anterior. Când este întâlnită **indent(10)** în expresia de ieşire, se execută a doua versiune pentru **indent()**, transmiţându-se valoarea 10 în parametrul **lungime**. Această versiune execută prima versiune, cu valoarea 10 din nou transmisă prin parametrul **lungime**. Procesul se repetă pentru fiecare apelare a funcţiei **indent()**.

Tipul parametrului pe care doriţi să îl aibă manipulatorul este sub controlul dvs. şi este determinat de tipul specificat ca al doilea parametru al funcţiei de manipulare. Acelaşi tip este apoi folosit când se creează clasa generică pentru manipulare. De exemplu, următorul program arată versiunea supraîncărcată a manipulatorului. De exemplu, următorul program arată cum se transmite o valoare **double** unei funcţii de manipulare. Se obţine apoi valoarea folosindu-se un format dolari-şi-cenţi.

```
#include <iostream.h>
#include <iomanip.h>

ostream &dolari(ostream &stream, double cantitate)
{
```

```
    stream.setf(ios::showpoint);
    stream << "$" << setw(10)  << setprecision(2)
           << cantitate;

    return stream;
}

omanimp <double> dolari(double cantitate) {
    return omanimp<double> (dolari, cantitate);
}
main()
{
    cout << dolari(123.123456);
    cout << "\n" << dolari(10.0);
    cout << "\n"; << dolari(1234.23);
    cout << "\n" << dolari(0.0);

    return 0;
}
```

Şi manipulatorii de intrare pot să preia un parametru. Următorul program dezvoltă manipulatorul **daparola()** prezentat mai înainte. Această versiune preia un argument care specifică de câte încercări beneficiază utilizatorul pentru a introduce corect parola.

```
// Acest program foloseste un manipulator pentru a
// introduce o parola
#include <iostream.h>
#include <iomanip.h>
#include <string.h>
#include <stdlib.h>

char *parola="ImiplaceC++";
char par[80];

// Introduce o parola
istream &daparola(istream &stream, int incercari)
{
    do {
        cout << "Introduceti parola: ";
        stream >> par;
        if(!strcmp(parola, par)) return stream;
        cout << "\a"; // clopotel
        incercari--;
    } while(incercari>0);

    cout << "Toate incercarile eronate!\n";
    exit(1) ; // nu s-a introdus parola

    return stream;
}

imanip<int> daparola(int incercari) {
    return imanip<int>(daparola, incercari);
}

main()
{
    // ofera trei incercari pentru introducerea parolei
    cin >> daparola(3);
    cout << "Parola corecta\n";

    return 0;
}
```


Reţineţi că formatul este acelaşi ca şi pentru manipulatorii de ieşire, cu două excepţii: trebuie folosit streamul de intrare **istream** şi se specifică clasa **imanip**.

Pe măsură ce veţi lucra cu C++, veţi vedea că manipulatorii adaptaţi vă pot ajuta în crearea instrucţiunilor de I/O.

O notiţă despre vechea bibliotecă de clase pentru streamuri

Când a fost inventat C++, a fost creată o bibliotecă de clase de I/O mică şi puţin diferită. Această bibliotecă este definită în fişierul STREAM.H. Însă, o dată cu evoluţia limbajului C++, ea a fost înlocuită de biblioteca de I/O descrisă în carte. Majoritatea compilatoarelor de C++ mai admit încă biblioteca veche pentru streamuri din motive de compatibilitate cu programele vechi în C++. Dar, ar trebui ca, atunci când scrieţi programe noi, să folosiţi biblioteca modernă de I/O definită în IOSTREAM.H.

Capitolul 18

I/O cu fişiere în C++



Chiar dacă abordarea I/O din C++ formează un sistem integrat, operaţiile de I/O cu fişiere (mai precis, cele de I/O pe disc) sunt suficient de specializate pentru a fi considerate un caz aparte, cu cerinţele şi particularităţile sale. Aceasta se întâmplă, în parte, deoarece cel mai uzual fişier este cel de pe disc, iar acesta are caracteristici şi facilităţi pe care alte fişiere nu le au. Reţineţi, totuşi, că operaţiile de I/O pentru fişiere sunt un simplu caz special al sistemului de I/O şi că mare parte din materialul discutat în acest capitol se aplică, de asemenea, şi streamurilor conectate la alte tipuri de echipamente.

fstream.h şi clasele de fişiere

Pentru a efectua I/O cu fişiere, trebuie să includeţi în programul dvs. fişierul antet FSTREAM.H. El defineşte mai multe clase, printre care **ifstream**, **ofstream** şi **fstream**. Aceste clase sunt derivate din **istream** şi, respectiv, din **ostream**. Amintiţi-vă că **istream** şi **ostream** sunt derivate din **ios**, astfel încât **ifstream**, **ofstream** şi **fstream** au, de asemenea, acces la toate operaţiile definite de această clasă de bază (prezentată în capitolul precedent).

Deschiderea şi închiderea unui fişier

Un fişier se deschide în C++ legându-l de un stream. Înainte de a putea să deschideţi un fişier, trebuie, pentru început, să aveţi un stream. Există trei tipuri de streamuri: de intrare, de ieşire şi de intrare/ieşire. Pentru a crea un stream de intrare, trebuie să-l declaraţi ca fiind din clasa **ifstream**. Pentru a crea unul de ieşire, trebuie să-l declaraţi ca fiind din clasa **ofstream**. Streamurile care vor efectua atât operaţii de intrare cât şi de ieşire trebuie declarate ca fiind de clasa **fstream**. De exemplu, acest fragment creează un stream de intrare, unul de ieşire şi unul capabil atât de intrare cât şi de ieşire.

```
ifstream intra; // intrare
ofstream iese; // iesire

fstream inies; // intrare si iesire
```

O dată ce aţi creat streamul, o cale de a-l asocia unui fişier este de a folosi funcţia **open()**. Această funcţie este membru al fiecăreia dintre cele trei clase stream. Prototipul său este:

void open(const char **numefisier*, int *mod*, int *acces=filebuf::openprot*);

numefisier este, aici, un nume de fişier, care poate include specificarea căii de acces. Valoarea din *mod* determină modul de deschidere a fişierului. Modul poate avea una (sau mai multe) dintre aceste valori:

ios::app
ios::ate
ios::binary
ios::in
ios::nocreate
ios::noreplace
ios::out
ios::trunc

Puteţi combina două sau mai multe dintre aceste valori unindu-le prin OR logic. Să vedem ce semnificaţie au ele.

Includerea valorii **ios::app** determină ca toate ieşirile către acel fişier să fie adăugate la sfârşit. Această valoare poate fi folosită doar cu fişierele ce acceptă ieşiri. Includerea valorii **ios::ate** determină căutarea sfârşitului fişierului la deschiderea sa. Cu toate acestea, pentru **ios::ate** operaţiile de I/O pot avea loc oriunde în interiorul fişierului.

Implicit, fişierele se deschid în mod text. Valoarea **ios::binary** determină deschiderea unui fişier în mod binar. Când fişierul este deschis în mod text, au loc diverse modificări de caractere, cum ar fi conversia secvenţei început de rând / salt la linie nouă. Însă, când un fişier este deschis în mod binar, nu apar asemenea modificări. Orice fişier poate fi deschis în mod text sau binar, chiar dacă el conţine un text formatat sau date binare brute. Singura diferenţă priveşte transformarea caracterelor.

Valoarea **ios::in** specifică faptul că fişierul acceptă intrări, iar valoarea **ios::out** că acceptă ieşiri. Totuşi, crearea unui stream folosind **ifstream** implică intrări iar crearea unuia cu **ofstream** implică ieşiri, astfel că, în aceste cazuri, nu mai este necesar să introduceţi aceste valori.

Includerea valorii **ios::nocreate** determină ca funcţia **open()** să eşueze dacă fişierul nu există deja. Valoarea **ios::noreplace** determină ca funcţia **open()** să eşueze dacă fişierul există deja.

Valoarea **ios::trunc** determină distrugerea şi reducerea la zero a mărimii conţinutului unui fişier preexistent care are acelaşi nume cu cel specificat.



***NOTĂ:** Standardul ANSI C++ propus specifică faptul că tipul parametrului mod trebuie să fie **openmod**, care este o formă de întreg (în general, un **int**). În mod curent, majoritatea implementărilor indică pur şi simplu că tipul parametrului mod este **int**.*

Valoarea din *acces* determină modul în care se obţine accesul la fişier. Valoarea implicită este **filebuf::openprot**, care specifică un fişier normal. (**filebuf** este o clasă derivată din **streambuf**.) În majoritatea cazurilor veţi permite *accesul* ca mod implicit. Totuşi, va trebui să verificaţi în manualul compilatorului ce alte opţiuni sunt valabile pentru acest parametru în mediul dvs. de operare. De

exemplu, în mediile pentru rețele, parametrul **acces** specifică de obicei opțiunile de departajare a fișierelor.

Următorul fragment deschide un fișier de ieșire normal.

```
ofstream out;

out.open("test", ios::out);
```

Totuși, veți vedea rareori (sau niciodată) **open()** apelat așa cum se arată aici, deoarece parametrul *mod* are și valori implicite. Pentru **ifstream**, valoarea implicită pentru *mod* este **ios::in**, iar pentru **ofstream**, **ios::out**. De aceea, instrucțiunea anterioară va arăta, de obicei, astfel:

```
out.open("test"); // Au valoarea implicita pentru iesire
```

Pentru a deschide un fișier de intrare și de ieșire, trebuie să specificați pentru *mod* atât valoarea **ios::in** cât și **ios::out**, cum se arată în următorul exemplu. (În acest caz nu există nici o valoare implicită pentru *mod*.)

```
fstream streamulmeu;
streamulmeu.open("test", ios::in | ios::out);
```

Dacă **open()** eșuează, **streamulmeu** va fi zero. De aceea, înainte de a folosi un fișier, ar trebui să efectuați un test pentru a vă asigura că operația de deschidere a reușit. Puteți face aceasta folosind o instrucțiune de tipul:

```
if(!streamulmeu) {
    cout << "Nu pot deschide fisierul.\n";
    // trateaza eroarea
}
```

Chiar dacă este pe deplin corect să deschideți un fișier folosind funcția **open()**, de cele mai multe ori nu o veți face, deoarece clasele **ifstream**, **ofstream** și **fstream** au funcții constructor care deschid automat fișierul. Funcțiile constructor au aceiași parametri și activări implicite ca și funcția **open()**. De aceea, de obicei veți vedea un fișier deschis așa cum se arată aici:

```
ifstream streamulmeu("fisierulmeu"); // deschide fisierul
                                     // pentru intrare
```

După cum am mai spus, dacă, din diferite motive, fișierul nu poate fi deschis, valoarea variabilei stream asociate va fi zero. De aceea, indiferent dacă pentru a deschide un fișier folosiți o funcție constructor sau o apelare explicită a funcției **open()**, ar trebui să testați valoarea streamului pentru a vi se confirma faptul că fișierul s-a deschis efectiv.

Pentru a închide un fișier, folosiți funcția membru **close()**. De exemplu, pentru a închide fișierul legat de streamul numit **streamulmeu**, folosiți această instrucțiune:

```
streamulmeu.close();
```

Funcția **close()** nu preia nici un parametru și nu returnează nici o valoare.

Citirea și scrierea fișierelor de text

Este foarte ușor să citiți și să scrieți un fișier de text. Folosiți pur și simplu operatorii << și >> în același fel în care o faceți și pentru I/O de la consolă, doar că, în loc să folosiți **cin** și **cout**, îi veți înlocui cu un stream care este legat de un fișier. De exemplu, acest program creează un fișier scurt pentru inventariere, care conține numele fiecărui articol și prețul său:

```
#include <iostream.h>
#include <fstream.h>

main()
{
    ofstream out("INVENTAR"); // iesire, fisier normal

    if(!out) {
        cout << "Nu pot deschide fisierul INVENTAR.\n";
        return 1;
    }

    out << "Radio " << 39.35 << endl;
    out << "Prajitor " << 19.95 << endl;
    out << "Mixer " << 24.80 << endl;

    out.close();
    return 0;
}
```

Următorul program citește fișierul de inventariere creat de programul anterior și îi afișează conținutul pe ecran.

```
#include <iostream.h>
#include <fstream.h>
```

```
main()
{
    ifstream in("INVENTAR"); // intrare
    if(!in) {
          cout << "Nu pot deschide fisierul INVENTAR.\n";
          return 1;
    }

    char articol[20];
    float pret;
    in >> articol >> pret;
    cout << articol << " " << pret << "\n";
    in >> articol >> pret;
    cout << articol << " " << pret << "\n";
    in >> articol >> pret;
    cout << articol << " " << pret << "\n";

    in.close();
    return 0;
}
```

Într-un fel, citirea şi scrierea fişierelor cu >> şi << este asemănătoare funcţiilor **fprintf()** şi **fscanf()** din C. Toate informaţiile sunt memorate în fişier în acelaşi format în care ar fi fost afişate pe ecran.

Urmează un alt exemplu de I/O pe disc. Acest program citeşte şiruri introduse de la tastatură şi le scrie pe disc. Programul se opreşte când utilizatorul introduce o linie goală. Pentru a folosi programul, specificaţi pe linia de comandă numele fişierului de ieşire.

```
#include <iostream.h>
#include <fstream.h>
#include <stdio.h>

main(int argc, char *argv[])
{
    if(argc!=2) {
          cout << "Utilizare: iesire <numefisier>\n";
          return 1;
    }

    ofstream out(argv[1]); // iesire, fisier normal

    if(!out) {
```



```
          cout << "Nu pot deschide fisierul de iesire.\n";
          return 1;
    }

    char sir[80];
    cout << "Scrie siruri pe disc, RETURN pentru a opri
              programul.\n";

    do {
          cout << ": ";
          gets(sir);
          out << sir << endl;
    }     while (*sir);

    out.close();
    return );
}
```

Când citiţi fişiere de tip text folosind operatorul >>, reţineţi că vor apărea anumite modificări de caractere. De exemplu, sunt omise caracterele de spaţii libere. Dacă doriţi să eliminaţi orice astfel de modificare, trebuie să folosiţi funcţiile de I/O binare din C++, discutate în paragraful următor.

Dacă, atunci când introduceţi, se întâlneşte un sfârşit de fişier, streamul legat de acel fişier va fi zero. (Acest lucru va fi ilustrat în următorul paragraf.)

I/O de tip binar

Există două feluri de a scrie şi de a citi datele de tip binar dintr-un fişier. Ambele metode vor fi explicate în continuare.



REŢINEŢI: Dacă veţi efectua operaţii binare asupra unui fişier, aveţi grijă să-l deschideţi folosind modul de specificare ***ios::binary****. Chiar dacă funcţiile fişierelor de tip binar vor lucra cu fişierele deschise în modul text, vor apărea unele modificări de caractere, care neagă scopul operaţiilor asupra fişierelor de tip binar.*

get() şi put()

O cale prin care puteţi să citiţi şi să scrieţi date binare este utilizarea funcţiilor membre **get()** şi **put()**. Aceste funcţii sunt orientate pe octeţi, ceea ce înseamnă că **get()** va citi un octet de date, iar **put()** va scrie un octet de date. Funcţia **get()** are

multe forme, dar aici este prezentată, împreună cu **put()**, cea mai des folosită versiune:

istream &get(char &*ch*);
ostream &put(char *ch*);

Funcția **get()** citește un singur caracter din streamul asociat și memorează valoarea în *ch*. Ea returnează o referință către stream. Funcția **put()** scrie *ch* în stream și returnează streamului o referință.

Următorul program afișează pe ecran conținutul unui fișier. El folosește funcția **get()**:

```
#include <iostream.h>
#include <fstream.h>

main(int argc, char *argv[])
{
    char ch;

    if(argc!=2) {
          cout << "Utilizare: PR <numefisier>\n";
          return 1;
    }

    ifstream in(argv[1], ios::in | ios::binary);
    if(!in) {
          cout << "Nu pot deschide fisierul. ";
          return 1;
    }
    while(in) { // cand se ajunge la sfarsit de fisier -
                // EOF - in va fi 0
          in.get(ch);
          cout << ch;
    }

    return 0;
}
```

După cum s-a spus în paragraful precedent, când se ajunge la sfârșitul fișierului, streamul asociat fișierului devine zero. De aceea, când **in** ajunge la sfârșit, streamul va fi zero, determinând oprirea buclei **while**.

Există, de fapt, un mod mai compact de a scrie bucla care citește și afișează un fișier, așa cum se prezintă aici:


```
while(in.get(ch))
    cout << ch;
```

Aceasta va lucra deoarece **get()** returnează o referință către streamul **in**, iar acesta va fi zero când se va întâlni sfârșitul fișierului.

Următorul program folosește **put()** pentru a scrie toate caracterele de la zero la 255 într-un fișier numit CHARS. După cum probabil știți, caracterele ASCII ocupă doar aproximativ jumătate din valorile care pot fi păstrate într-un **char**. Celelalte valori sunt, de obicei, numite *set de caractere extins* și includ printre altele caractere din limbi străine și simboluri matematice. (Nu toate sistemele admit setul de caractere extins, dar majoritatea o fac.)

```
#include <iostream.h>
#include <fstream.h>
main()
{
    int i;
    ofstream out("CHARS", ios::out | ios::binary);

    if(!out) {
          cout << "Nu pot deschide fisierul.\n";
          return 1;
    }

    // scrie toate caracterele pe disc
    for(i=0; i<257; i++) out.put((char) i);
    out.close();
    return 0;
}
```

Poate că veți găsi interesantă ideea să verificați conținutul fișierului CHARS pentru a vedea ce caractere extinse are calculatorul dvs.

read() și write()

A doua cale de a citi și a scrie blocuri de date în binar este folosirea funcțiilor din C++ **read()** și **write()**. Prototipurile lor sunt:

istream &read(unsigned char **buf*, int *numar*);
ostream &write(const unsigned char **buf*, int *numar*);

Funcția **read()** citește *numar*-ul de octeți din streamul asociat și îi pune în bufferul indicat de *buf*. Funcția **write()** scrie în streamul asociat *numar*-ul de octeți citiți din bufferul spre care indică *buf*.



NOTĂ: *Standardul propus pentru ANSI C++ specifică tipul parametrului numar ca fiind* ***streamsize****, care este un* ***typedef*** *pentru un tip de întreg. În mod curent majoritatea compilatoarelor de C++ indică pur şi simplu că numar este un întreg, aşa cum se arată în prototipurile anterioare. În general, propunerea de standard ANSI C++ foloseşte* ***streamsize*** *ca tip al oricărui obiect care specifică numărul de octeţi transferaţi printr-o operaţie de intrare sau de ieşire.*

Următorul program scrie o structură pe disc, apoi o citeşte.

```
#include <iostream.h>
#include <fstream.h>
#include <string.h>

struct stare {
    char nume[80];
    float bilant;
    unsigned long numar_cont;
};

main()
{
    struct stare ct;

    strcpy(ct.nume, "Ralph Trantor");
    ct.bilant = 1123.23;
    ct.numar_cont = 34235678;

    ofstream outbal("bilant", ios::out | ios::binary);

    if(!outbal) {
        cout << "Nu pot deschide fisierul.\n";
        return 1;
    }

    outbal.write((unsigned char *) &ct, sizeof(struct stare));
    outbal.close();

    // acum, citeste ce a scris
    ifstream inbal("bilant", ios::in | ios::binar);

    if(!inbal) {
        cout << "Nu pot deschide fisierul.\n";
        return 1;
    }

    inbal.read((unsigned char *) &ct, sizeof(struct stare));

    cout << ct.nume << endl;
    cout << "Cont # " << ct.numar_cont;
    cout.precision(2);
    cout.setf(ios::fixed);
    cout << endl << "Bilant: $" << ct.bilant;

    inbal.close();
    return 0;
}
```


După cum puteţi vedea, este necesară o singură apelare a funcţiilor **read()** şi **write()** pentru a citi sau scrie întreaga structură. Nu este nevoie ca fiecare câmp individual să fie citit sau scris separat. După cum ilustrează acest exemplu, bufferul poate fi orice tip de obiect.



NOTĂ: *Când* ***read()*** *şi* ***write()*** *operează asupra unui buffer care nu este definit ca o matrice de caractere, în interiorul apelărilor lor, sunt necesari modelatorii de tip. Datorită stricteţii de verificare a tipului din C++ un pointer de un anumit tip nu va fi convertit automat într-unul de alt tip.*

Dacă se ajunge la sfârşitul fişierului înainte de a fi citite *numar* de caractere, atunci **read()** se opreşte iar bufferul conţine atâtea caractere câte a înregistrat. Puteţi şti câte caractere au fost citite folosind o altă funcţie membru, numită **gcount()**, care are următorul prototip:

```
int gcount();
```

Ea returnează numărul de caractere citite de ultima operaţie de intrare binară. Următorul program arată un alt exemplu de folosire a funcţiilor **read()** şi **write()** şi ilustrează utilizarea lui **gcount()**.

```
#include <iostream.h>
#include <fstream.h>

main(void)
{
    float fnum[4] = {99.75, -34.4, 1776.0, 200.1};
    int i;
```

```
    ofstream out("numere", ios::out | ios::binary);
    if(!out) {
        cout << "Nu pot deschide fisierul.\n";
        return 1;
    }

    out.write((unsigned char *) &fnum, sizeof fnum);

    out.close();

    for(i=0; i<4; i++) // sterge matricea
        fnum[i] = 0.0;

    ifstream in("numere", ios::in | ios::binary);
    in.read((unsigned char *) &fnum, sizeof fnum);

    // afla cati octeti au fost cititi
    cout << in.gcount() << " octeti cititi\n";

    for(i=0; i<4; i++) // arata valorile citite din fisier
    cout << fnum[i] << " ";

    in.close();

    return 0;
}
```

Acest program scrie pe disc o matrice de valori în virgulă mobilă şi apoi o citeşte. După apelarea funcţiei **read()**, **gcount()** este folosită pentru a determina câţi octeţi au fost citiţi efectiv.

Mai multe funcţii get()

În plus faţă de forma prezentată mai devreme, funcţia **get()** este supraîncărcată în mai multe feluri. Iată prototipurile celor mai des utilizate forme de supraîncărcare:

istream &get(char **buf*, int *numar*, char *delim*='\n');
int get();

Prima formă de supraîncărcare citeşte caractere dintr-o matrice spre care indică *buf*, până când ori se atinge *număr* de caractere, ori se întâlneşte caracterul specificat de *delim*. **get()** va termina cu null matricea spre care indică *buf*. Dacă nu se specifică nici un parametru pentru *delim*, rolul de limitator va fi jucat de caracterul de linie nouă. Dacă, în streamul de intrare, este întâlnit caracterul de delimitare, el *nu* este extras şi el rămâne în stream până la următoarea operaţie de intrare.

A doua formă de supraîncărcare pentru **get()** returnează din stream caracterul următor. Ea returnează EOF dacă se întâlneşte sfârşitul fişierului. Această formă a funcţiei **get()** este similară funcţiei **get()** din C.

getline()

O altă funcţie membru care efectuează intrări este **getline()**. Prototipul ei este:

istream &getline(char **buf*, int *numar*, char *delim*='\n');

După cum puteţi vedea, această funcţie este practic identică cu versiunea **get()**, **get(buf, num, delim)**. Ea citeşte caractere de la intrare şi le pune într-o matrice spre care indică *buf* până când ori se citesc *numar* de caractere, ori se întâlneşte caracterul specificat de *delim*. Dacă nu este specificat, *delim* este implicit caracterul de linie nouă. Matricea spre care indică *buf* se termină cu null. Diferenţa dintre **get(buf, num, delim)** şi **getline()** este că ultima citeşte şi extrage delimitatorul din streamul de intrare.

Iată un program care ilustrează funcţia **getline()**. El citeşte conţinutul unui fişier text, câte o linie o dată, şi îl afişează pe ecran.

```
// Citeste si afiseaza un fisier text linie cu linie.
#include <iostream.h>
#include <fstream.h>

main(int argc, char *argv[])
{
    if(argc!=2) {
        cout << "Utilizare: Afiseaza<numefisier>\n";
        return 1;
    }
    ifstream in(argv[1]);
    if(!in) {
        cout << "Nu pot deschide fisierul.\n";
        return 1;
    }

    char sir[255];

    while(in) {
        in.getline(sir, 255); // limita implicita '\n'
```

```
        cout << sir << endl;
    }

    in.close();
    return 0;
}
```

Detectarea EOF

Puteți să detectați când s-a ajuns la sfârșitul fișierului folosind funcția membru **eof()**, care are acest prototip:

int eof();

Ea returnează o valoare nonzero când a fost atins sfârșitul fișierului; altfel, returnează zero.

*NOTĂ: Propunerea pentru standardul ANSI C++ specifică tipul returnat de **eof()** ca fiind **bool**. Totuși, cele mai uzuale compilatoare de C++ disponibile nu admit tipul de dată **bool**. Din punct de vedere practic nu are importanță faptul că tipul returnat de **eof** este specificat ca **bool** sau ca **int** deoarece **bool** este ridicat în orice expresie la rang de **int**.*

Următorul program folosește **eof()** pentru a afișa conținutul unui fișier atât în hexazecimal cât și în cod ASCII.

```
/* Display contents of specified file
   in both ASCII and hex.
*/
#include <iostream.h>
#include <fstream.h>
#include <ctype.h>
#include <iomanip.h>
#include <stdio.h>

main(int argc, char *argv[])
{
    if(argc(!=2) {
        cout << "Usage: Display <filename>\n";
        return 1;
    }
```



```
    ifstream in(argv[1], ios::in | ios::binary);

    if(!in) {
        cout << "Nu pot deschide fisierul.\n";
        return 1;
    }

    register int i, j;
    int count = 0;
    char c[16];

    cout.setf(ios::uppercase);
    while(!in.eof()) {
        for(i=0; i<16 && !in.eof(); i++) {
            in.get(c[i]);
        }
        if(i<16) i--; // get rid of eof

        for(j=0; j<i; j++)
            cout << setw(3) << hex  << (int) c[j];
        for(; j<16; j++) cout << " ";
        cout << "\t";
        for(j=0; j<i; j++)
            if(isprint(c[j])) cout << c[j];
            else cout << ".";

        cout << endl;

        count++;
        if(count==16) {
            count = 0;
            cout << "Press ENTER to continue: ";
            cin.get();
            cout << endl;
        }
    }

    in.close;
    return 0;
}
```

Când acest program este folosit pentru a se afișa pe sine însuși, primul ecran arată astfel:

```
2F 2A 20 44 69 73 70 6C 61 79 20 63 6F 6E 74 65   /* Display conte
6E 74 73 20 6F 66 20 73 70 65 63 69 66 69 65 64   nts of specified
20 66 69 6C 65  D  A 20 20 20 69 6E 20 62 6F 74    file..   in bot
68 20 41 53 43 49 49 20 61 6E 64 20 69 6E 20 68   h ASCII and in h
65 78 2E  D  A 2A 2F  D  A 23 69 6E 63 6C 75 64   ex...*/..#includ
65 20 3C 69 6F 73 74 72 65 61 6D 2E 68 3E  D  A   e <iostream.h>..
23 69 6E 63 6C 75 64 65 20 3C 66 73 74 72 65 61   #include <fstrea
6D 2E 68 3E  D  A 23 69 6E 63 6C 75 64 65 20 3C   m.h>..#include <
63 74 79 70 65 2E 68 3E  D  A 23 69 6E 63 6E 75   ctype.h> ..#inclu
64 65 20 3C 69 6F 6D 61 6E 69 70 2E 68 3E  D  A   de <iomanip.h>..
23 69 6E 63 6C 75 64 65 20 3C 73 74 64 69 6F 2E   #include <stdio.
68 3E  D  A  D  A 6D 61 69 6E 28 69 6E 74 20 61   h>....main(int a
72 67 63 2C 20 63 68 61 72 20 2A 61 72 67 76 5B   rgc, char *argv[
5D 29  D  A 7B  D  A 20 20 69 66 28 61 72 67 63   ])..{..  if(argc
21 3D 32 29 20 7B  D  A 20 20 20 20 63 6F 75 74   !=2) {..     cout
20 3C 3C 20 22 55 73 61 67 65 3A 20 44 69 73 70    << "Usage: Disp
Press ENTER to continue:
```

Funcţia ignore()

Puteţi să folosiţi funcţia membru **ignore()** pentru a citi şi a ignora caractere din streamul de intrare. Ea are prototipul acesta:

istream &ignore(int *num*=1, int *delim*=EOF);

Ea citeşte şi elimină caracterele până când sunt ignorate *numar* de caractere (implicit, 1) sau până când se întâlneşte caracterul specificat prin *delim* (implicit, EOF). Dacă se întâlneşte caracterul de delimitare, el nu este extras din streamul de intrare.

Următorul program citeşte un fişier numit TEST. El ignoră caracterele până când se întâlneşte un spaţiu sau până când au fost citite 10 caractere. Apoi afişează restul fişierului.

```
#include <iostream.h>
#include <fstream.h>

main()
{
    ifstream in("test");

    if(!in) {
```



```
        cout << "Nu pot deschide fisierul.\n";
        return 1;
    }

    /* Ignora pana la 10 caractere sau pana este gasit
       primul spatiu. */
    in.ignore(10, ' ');
    char c;
    while(in) {
        in.get(c);
        cout << c;
    }

    in.close();
    return 0;
}
```

peek() şi putback()

Folosind **peek()**, puteţi să obţineţi caracterul următor din streamul de intrare, fără să-l extrageţi de aici. Funcţia are următorul prototip:

int peek();

Ea returnează următorul caracter din stream sau EOF dacă s-a ajuns la sfârşitul fişierului.

Folosind **putback()**, puteţi să returnaţi ultimul caracter citit dintr-un stream înapoi în acelaşi stream. Prototipul ei este:

istream &putback(char *c*);

unde *c* este ultimul caracter citit.

flush()

Când se efectuează ieşirea datelor, ele nu sunt neapărat scrise imediat la echipamentul fizic de care este legat streamul. Informaţiile pot fi memorate într-un buffer intern până când acesta se umple. Doar atunci conţinutul bufferului este scris pe disc. Însă, folosind **flush()**, puteţi forţa ca informaţiile să fie scrise efectiv pe disc înainte ca bufferul să fie plin. Prototipul ei este:

ostream &flush();

Apelările funcţiei **flush()** sunt de dorit atunci când programul rulează într-un mediu neprietenos (de exemplu, în situaţii în care apar frecvent întreruperi de curent).

NOTĂ: Închiderea unui fişier sau terminarea normală a unui program determină, de asemenea, golirea tuturor bufferelor.

Accesul aleator

În sistemul de I/O din C++, accesul aleator se efectuează folosind funcţiile **seekg()** şi **seekp()**. Formele lor cele mai uzuale sunt:

istream &seekg(streamoff *offset*, seek_dir *origine*);
ostream &seekp(streamoff *offset*, seek_dir *origine*);

streamoff este un tip definit în IOSTREAM.H, capabil să conţină cea mai mare valoare validă pe care o poate avea *offset*, iar **seek_dir** este o enumerare care are aceste valori:

ios::beg
ios::cur
ios::end

Sistemul de I/O din C++ operează cu doi pointeri asociaţi unui fişier. Unul este *pointerul get*, care specifică unde va apărea următoarea operaţie de intrare în fişier. Celălalt este *pointerul put*, şi specifică unde va avea loc în fişier următoarea operaţie de ieşire. După fiecare operaţie de intrare sau de ieşire, pointerul corespunzător este avansat secvenţial, automat. Dar folosirea funcţiilor **seekg()** şi **seekp()** vă permite să aveţi acces la fişier într-un mod nesecvenţial.

Funcţia **seek()** deplasează pointerul get al fişierului asociat cu un număr de octeţi egal cu valoarea *offset* faţă de *originea* specificată, care trebuie să aibă una dintre aceste trei valori:

ios::beg	Început de fişier
ios::cur	Locaţie curentă
ios::end	Sfârşit de fişier

Funcţia **seekp()** deplasează pointerul put al fişierului asociat cu un număr de octeţi egal cu valoarea *offset* faţă de *originea* specificată, care trebuie să aibă una dintre valorile de mai sus.

Următorul program ilustrează funcţia **seekp()**. El vă permite să modificaţi un anumit caracter dintr-un fişier. Specificaţi pe linia de comandă un nume de fişier, urmat de numărul octetului din fişier pe care doriţi să-l modificaţi, urmat de noul caracter. Observaţi că fişierul este deschis pentru operaţii de citire/scriere.


```
#include <iostream.h>
#include <fstream.h>
#include <stdlib.h>

main(int argc, char *argv[])
{
    if(argc!=4) {
        cout << "Utilizare: MODIFICA <numefisier> <octet>
                <caracter>\n";
        return 1;
    }
 fstream out(argv[1], ios::in | ios::out | ios::binary);
    if(!out){
    cout << "Nu pot deschide fisierul.\n";
    return 1;
 }

 out.seekp(atoi(argv[2]), ios::beg);

    out.put (*argv[3]);
    out.close();

    return 0;
 }
```

De exemplu, pentru a înlocui prin intermediul acestui program cu **Z** al 12-lea octet din fişierul numit TEST, folosiţi această linie de comandă:

change test 12 Z

Următorul program foloseşte **seekg()**. El afişează conţinutul unui fişier începând cu locaţia pe care o specificaţi pe linia de comandă.

```
#include <iostream.h>
#include <fstream.h>
#include <stdlib.h>

main(int argc, char *argv[])
{
    char ch;
```

```
    if(argc!=3) {
        cout << "Utilizare: ARATA <numefisier> <pozitia
                initiala>\n";
        return 1;
    }

    ifstream in(argv[1], ios::in | ios::binary);
    if(!in) {
        cout << "Nu pot deschide fisierul.";
        return 1;
    }

    in.seekg(atoi(argv[2]), ios::beg);

    while(in.get(ch))
        cout << ch;

    return 0;
}
```

Următorul program foloseşte atât **seekp()** cât şi **seekg()** pentru a inversa primele <numar> de caractere din fişier.

```
#include <iostream.h>
#include <fstream.h>
#include <stdlib.h>

main(int argc, char *argv[])
{
    if(argc!=3) {
        cout << "Utilizare: Invers <numefisier> <numar>\n";
        return 1;
    }

    fstream inout(argv[1], ios::in | ios::out | ios::binary);
    if(!inout) {
        cout << "Nu pot deschide fisierul.\n";
        return 1;
    }

    long e, i, j;
    char c1, c2;
    e = atol(argv[2]);
```



```
    for(i=0, j=e; i<j; i++, j--) {
        inout.seekg(i, ios::beg);
        inout.get(c1);
        inout.seekg(j, ios::beg);
        inout.get(c2);

        inout.seekp(i, ios::beg);
        inout.put(c2);
        inout.seekp(j, ios::beg);
        inout.put(c1);
    }

    inout.close();
    return 0;
}
```

Pentru a folosi programul, specificaţi numele fişierului în care doriţi să faceţi inversările, urmat de numărul de caractere de inversat. De exemplu, pentru a inversa primele zece caractere din fişierul numit TEST, folosiţi această linie de comandă:

reverse test 10

Dacă fişierul conţine

Acesta e un test;

după executarea programului el va conţine următoarele:

u e atsecAn test.

Obţinerea poziţiei curente dintr-un fişier

Folosind funcţiile următoare, puteţi determina poziţia curentă a fiecărui pointer al fişierului:

streampos tellg();
streampos tellp();

streampos este, aici, un tip definit în IOSTREAM.H care este capabil să păstreze cea mai mare valoare pe care o poate returna oricare dintre cele două funcţii.

Puteți folosi valorile returnate de **tellg()** și de **tellp()** ca argumente pentru următoarele forme de **seekg()** și, respectiv, **seekp()**:

istream &seekg(streampos *poz*);
ostream &seekp(streampos *poz*);

Aceste funcții vă permit să salvați poziția curentă a fișierului, să efectuați alte operații specifice fișierelor și apoi să reveniți în locația salvată anterior.

Starea de I/O

Sistemul de I/O din C++ întreține informații de stare relativ la rezultatul fiecărei operații de I/O. Starea curentă a sistemului de I/O este păstrată într-un întreg în care sunt codificați următorii indicatori:

Nume	Semnificație
eofbit	1, când se întâlnește sfârșitul fișierului 0, altfel
failbit	1, când apare o eroare de I/O(posibil nu fatală) 0, altfel
badbit	1, când apare o eroare fatală 0, altfel

Acești indicatori sunt enumerați în cadrul clasei **ios**. De asemenea, în **ios** mai este definit și **goodbit** (fără eroare), care are valoarea 0.

Există două căi prin care puteți obține informații de stare pentru I/O. Mai întâi, puteți apela funcția membru **rdstate()**. Ea are prototipul acesta:

int rdstate();

Ea returnează starea curentă a indicațiilor de eroare codificați într-un întreg. După cum probabil bănuiți urmărind lista precedentă de indicatori, **rdstate()** returnează zero când nu apare nici o eroare. Altfel, se activează anumiți biți de eroare.

*NOTĂ: Standardul ANSI C++ propus specifică tipul returnat de **rdstate()** ca fiind **iostate**, care este un **typdef** pentru o anumită formă de întreg. Uzual, majoritatea compilatoarelor de C++ indică tipul returnat de **rdstate** ca fiind **int**.*

Următorul program ilustrează **rdstate()**. El afișează conținutul unui fișier text. Dacă apare o eroare, programul o anunță, folosind **verifstare()**.


```
#include <iostream.h>
#include <fstream.h>

void verifstare(ifstream &in);

main(int argc, char *argv[])
{
    if(argc!=2) {
          cout << "Utilizare: Afiseaza <numefisier>\n";
          return 1;
    }

    ifstream in(argv[1]);
    if(!in) {
          cout << "Nu pot deschide fisierul.\n";
          return 1;
    }
    char c;
    while(in.get(c)) {
          cout << c;
          verifstare(in);
    }

    verifstare(in); // verifica starea finala

    in.close();
    return 0;
}

void verifstare(ifstream &in)
{
    int i;

    i = in.rdstate();
    if(i & ios::eofbit)
          cout << "S-a intalnit EOF\n";
    else if(i & ios::failbit)
          cout << "Eroare I/O nefatala\n";
    else if(i & ios::badbit)
          cout << "Eroare fatala\n";
}
```

Acest program va semnala întotdeauna o singură „eroare". După terminarea buclei **while**, apelarea finală pentru **verifstare()** semnalează, după cum ne

aşteptam, că a fost întâlnit EOF. Poate că veţi găsi utilă funcţie **verifstare()** în programele pe care le scrieţi.

Cealaltă cale prin care puteţi verifica dacă a apărut o eroare este folosirea uneia sau mai multora dintre aceste funcţii:

```
int bad();
int eof();
int fail();
int good();
```

Funcţia **eof()** a fost discutată mai devreme. Funcţia **bad()** returnează adevărat dacă este activat **badbit**. Funcţia **fail()** returnează adevărat dacă **failbit** este activat. Funcţia **good()** returnează adevărat dacă nu există erori. În caz contrar ele returnează fals.

*NOTĂ: Standardul ANSI C++ propus specifică tipul returnat de **bad(), eof(), fail()** şi **good()** ca **bool**. Dar compilatoarele uzuale specifică tipul returnat de funcţii ca fiind **int**. Din punct de vedere practic, diferenţa este nesemnificativă deoarece, într-o expresie, **bool** este convertit automat în **int**.*

O dată ce a apărut o eroare, ea trebuie să fie îndepărtată înainte ca programul să continue. Pentru a face aceasta, folosiţi funcţia **clear()**, care are următorul prototip:

void clear(int *indicatori*=0);

Dacă *indicatori* este 0 (aşa cum este implicit), toţi indicatorii pentru erori sunt şterşi (iniţializaţi cu zero). Altfel, stabiliţi ca *indicatori* valorile pe care doriţi să le ştergeţi sau indicatorii.

I/O şi fişiere adaptate

În **Capitolul 17** aţi învăţat cum să supraîncărcaţi operatorii de inserţie şi de extracţie relativ la clasele dvs. proprii. În acel capitol au fost efectuate doar I/O pentru consolă. Totuşi, deoarece toate streamurile C++ sunt identice, puteţi folosi aceeaşi funcţie pentru inserţie supraîncărcată pentru ieşiri pe ecran sau într-un fişier, fără nici o modificare. Ca ilustrare, acest program modifică exemplul agendei de telefon din **Capitolul 17**, astfel încât să memoreze o listă pe disc. Programul este foarte simplu: el vă permite să adăugaţi nume pe listă sau să afişaţi lista pe ecran. Totuşi, ca un exerciţiu, poate că veţi găsi că este interesant să dezvoltaţi programul astfel încât să găsească un anumit număr şi să şteargă numere nedorite.


```
#include <iostream.h>
#include <fstream.h>
#include <string.h>

class agendatelefon {
   char nume[80];
   char codzona[4];
   char prefix[4];
   char numar[5];
public:
   agendatelefon() { };
   agendatelefon(char *n, int *a, int *p, int *nm)
   {
         strcpy(nume, n);
         strcpy(codzona, a);
         strcpy(prefix, p);
         strcpy(numar, nm);
   }
   friend ostream &operator<<(ostream &stream,
                                     agendatelefon o);
   friend istream &operator>>(istream &stream,
                                     agendatelefon &o);
};

// Afiseaza numele si numarul de telefon.
ostream &operator<<(ostream &stream, agendatelefon o)
{
   stream << o.nume << " ";
   stream << "(" << o.codzona << ") ";
   stream << o.prefix << "-";
   stream <<o.num << "\n";
   return stream; // trebuie sa returneze stream
}
// Introduce nume si numere de telefon.
istream &operator>>(istream &stream, agenda telefon &o)
{
   cout << "Introduceti nume: ";
   stream >> o.nume;
   cout << "Introduceti codzona: ";
   stream >> o.codzona;
   cout << "Introduceti prefix: ";
   stream >> o.prefix;
   cout << "Introduceti numar: ";
```

```
    stream >> o.numar;
    cout << "\n";
    return stream;
}

main()
{
    agendatelefon a;
    char c;

    fstream at("telefon", ios::in | ios::out | ios::app);
    if(!at) {
          cout << "Nu pot deschide fisierul agenda de
                    telefon.\n";
          return 1;
    }

    for(;;) {
          do {
                cout << "1. Introduceti numerele\n";
                cout << "2. Afiseaza numerele\n";
                cout << "3. Parasiti programul\n";
                cout << "\nIntroduceti o optiune: ";
                cin >> c;
          }     while(c<'1' || c>'3');
          switch(c) {
                case '1':
                      cin >> a;
                      cout << "Intrarea este: ";
                      cout << a; // afiseaza pe ecran
                      at << a; // scrie pe disc
                      break;
                case '2';
                      char ch;
                      at.seekg(0, ios::beg);
                      while(!at.eof()) {
                            at.get(ch);
                            cout << ch;
                      }
                      at.clear(); // reset eof
                      cout << endl;
                      break;
                case '3':
```

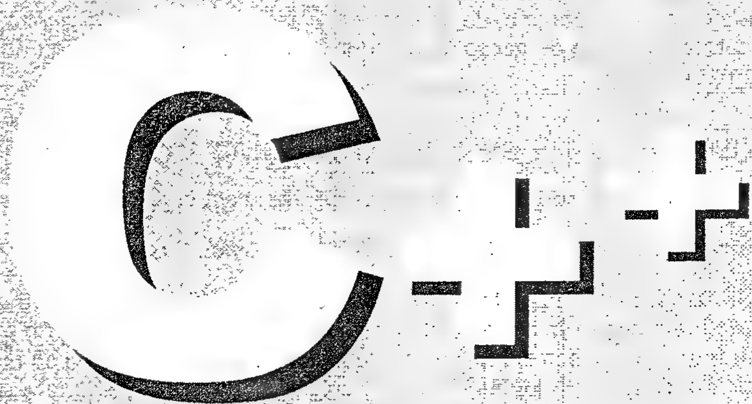


```
                      at.close();
                      return 0;
          }
    }
}
```

Reţineţi că operatorul suprapus << poate fi folosit pentru a scrie într-un fişier de pe disc sau pe ecran, fără nici o schimbare. Aceasta este una dintre cele mai importante şi folositoare facilităţi ale abordării de I/O din C++.

Capitolul 19

I/O bazate pe matrice



În plus față de I/O pentru consolă și fișier, sistemul de I/O din C++ axat pe stream permite operații de I/O *bazate pe matrice*. Operațiile de I/O *bazate pe matrice* folosesc memoria RAM ca dispozitiv de intrare, de ieșire sau și una și alta. Ele sunt efectuate prin streamuri C++ normale. De fapt, toate informațiile prezentate în cele două capitole precedente sunt aplicabile și pentru operațiile de I/O bazate pe matrice. Ceea ce diferă este că dispozitivul care se leagă de stream este memoria.

O parte din literatura de C++ numește operațiile de I/O bazate pe matrice *I/O cu RAM*. De asemenea, deoarece streamurile sunt, ca și toate streamurile de C++, capabile să manevreze informații formatate, operațiile de I/O bazate pe matrice sunt numite și *formatare în RAM*. (Uneori se mai folosește și termenul arhaic *formatare incore*. Totuși, deoarece memoria „core" este de domeniul trecutului, această carte va folosi termenul *formatare în RAM* și *bazată pe matrice*.)

Operațiile de I/O din C++ bazate pe matrice au același efect ca funcțiile **sprintf()** și **sscanf()** din C. Ambele metode folosesc memoria ca pe un dispozitiv de intrare și de ieșire.

Pentru a folosi în programele dvs. operații de I/O bazate pe matrice, trebuie să includeți STRSTREA.H.

Clasele bazate pe matrice

Clasele de I/O bazate pe matrice sunt **istrstream**, **ostrstream** și **strstream**. Aceste clase sunt folosite pentru a crea streamuri de intrare, de ieșire și, respectiv, de intrare/ieșire. Toate aceste clase au **strstreambuf** drept una dintre clasele de bază. Acesta definește foarte multe detalii de lucru la nivel scăzut, care sunt folosite și de clasele derivate. În plus față de **strstreambuf**, clasa **istrstream** are, de asemenea, **istream** ca bază. Clasa **ostrstream** este derivată și din **ostream**, iar clasa **strstream** conține, de asemenea, clasele **iostream**. De aceea, toate clasele bazate pe matrice au acces la aceleași funcții membre la care au acces și clasele „normale" de I/O.

Crearea unui stream de ieșire bazat pe matrice

Pentru a lega un stream la o matrice, folosiți acest constructor **ostrstream**:

ostrstream ostr(char **buffer*, int *marime*, int *mod=ios::out*)

buffer este, aici, un pointer spre o matrice care va fi folosită pentru a păstra caracterele scrise în stream. Mărimea matricei este transmisă prin parametrul *size*. Streamul este deschis implicit pentru ieșire normală, dar puteți să îl uniți prin OR logic de diverse alte opțiuni pentru a crea modul care vă este necesar. (De exemplu, puteți să includeți **ios::app** pentru a determina ca ieșirea să fie scrisă la sfârșitul informațiilor conținute deja în matrice.) În cele mai multe cazuri, *mod* va fi lăsat la valoarea implicită.

O dată ce ați deschis un stream de ieșire bazat pe matrice, toate ieșirile din acest stream sunt trecute într-o matrice. Totuși, nici o ieșire nu va fi scrisă în afara limitelor acesteia. O încercare de a face așa ceva va determina o eroare.

Iată un program simplu, care ilustrează un stream de ieșire bazat pe matrice:

```
#include <strstream.h>
#include <iostream.h>

main()
{
    char sir[80];

    ostrstream iesiri(sir, sizeof(sir));

    iesiri << "Hello ";
    iesiri << 99-14 << hex << " ";
    iesiri.setf(ios::showbase);
    iesiri << 100 << ends;
    cout << sir; // afiseaza sirul la consola

    return 0;
}
```

Acest program afișează **Hello 85 0x64**. Rețineți că **iesiri** este un stream ca oricare altul, cu aceleași facilități. Singura diferență este că dispozitivul la care este legat este memoria. Deoarece **iesiri** este un stream, manipulatorii ca **hex** și **ends** sunt perfect valizi. De asemenea, este permisă utilizarea funcțiilor membre din **ostream**, cum ar fi **setf()**.

Dacă doriți să obțineți matrice terminate cu null, trebuie să scrieți null explicit. În programul precedent manipulatorul **ends** a fost folosit pentru a încheia șirul cu null, dar ați fi putut folosi și **'\0'**.

Dacă nu sunteți foarte sigur de ce se întâmplă efectiv în programul precedent, comparați-l cu următorul program în C. Acesta este funcțional echivalent versiunii în C++, doar că, pentru a construi o matrice de ieșire, folosește **sprintf()**.

```
#include <stdio.h>

main()
{
    char sir[80];

    sprintf(sir, "Hello %d %#x", 99-14, 100);
    printf(sir);
```

```
    return 0;
}
```

Puteți să determinați câte caractere sunt în matricea de ieșire apelând funcția membru **pcount()**. Ea are următorul prototip:

int pcount();

Numărul returnat de **pcount()** include și null de încheiere, dacă acesta există.

Următorul program ilustrează **pcount()**. El spune că în **iesiri** sunt 17 caractere - 16 caractere plus caracterul null de încheiere.

```
#include <strstrea.h>
#include <iostream.h>

main()
{
    char sir[80];

    ostrstream iesiri(sir, sizeof(sir));

    iesiri << "Hello ";
    iesiri << 34 << " " << 1234.23;
    iesiri << ends; // null de terminare

    cout << iesiri; // afiseaza numarul de caractere
                    // din iesiri

    cout << " " << sir;

    return 0;
}
```

Utilizarea unei matrice ca intrare

Pentru a lega un stream de intrare de o matrice, folosiți acest constructor **istrstream**;

istrstream istr(char **buffer*);

buffer este, aici, un pointer spre o matrice care va fi folosită ca sursă de caractere de fiecare dată când se efectuează o intrare în stream. Conținutul matricei spre care indică *buffer* trebuie să fie încheiat cu null. Însă, acesta nu este citit niciodată.

Iată un exemplu care folosește ca intrare un șir:


```
#include <iostream.h>
#include <strstrea .h>

main()
{
    char s[] = "10 Hello 0x88 12.23 gata";

    istrstream ins(s);

    int i;
    char sir[80];
    float f;

    // citeste: 10 Hello
    ins >> i;
    ins >> sir;

    cout << i << " " << sir << endl;

    // citeste 0x88 12.23 gata
    ins >> i;
    ins >> f;
    ins >> str
    cout << hex << i << " " << f << " " << sir;

    return 0;
}
```

Dacă doriți ca doar o parte a șirului să fie folosită pentru intrare, utilizați această formă de constructor **istrstream**:

istrstream istr(char **buffer*, int *marime*);

Aici vor fi folosite doar primele *marime* elemente ale matricei spre care indică *buffer*. Acest șir nu trebuie să fie terminat cu null deoarece factorul care determină mărimea șirului este valoarea argumentului *marime*.

Streamurile legate de memorie se comportă exact la fel ca și cele legate la alte echipamente. De exemplu, următorul program ilustrează cum poate fi citit conținutul unei matrice de tip text. Când se ajunge la sfârșitul matricei (la fel ca pentru fișiere), **ins** va fi zero.

```
/* Acest program arata cum se citeste continutul oricarei
   matrice care contine un text. */
#include <iostream.h>
#include <strstream.h>

main()
{
    char s[] = "10.23 acesta este un text !#?@\n";

    istrstream ins(s);

    char ch;

    /* Aceasta va citi si va afisa continutul
       oricarei matrice tip text. */
    ins.unsetf(ios::skipws); // nu omite spatii
    while (ins) { // 0 cind se ajunge la sfarsitul matricei
          ins >> ch;
          cout << ch;
    }

    return 0;
}
```

Folosirea funcţiilor membre tip pentru streamuri bazate pe matrice

Streamurile bazate pe matrice pot, de asemenea, să fie accesibile prin intermediul funcţiilor membre standard din **ios**, cum ar fi **get()** şi **put()**. Starea streamului bazat pe matrice poate fi determinată cu funcţii ca **rdstate()**, **good()**, **bad()** ş.a.m.d. De asemenea, puteţi folosi **eof()** pentru a determina când se ajunge la sfârşitul matricei. De exemplu, următorul program determină cum se citeşte conţinutul unei matrice folosindu-se **get()**.

```
#include <iostream.h>
#include <strstrea .h>

main()
{
    char s[] = "abcdefghijklmnop";

    istrstream ins();

    char ch;

    // Aceasta va citi continutul oricarui tip de matrice.
    while (!ins.eof()) {
          ins.get(ch);
          cout << ch;
    }

    return 0;
}
```


Funcţiile membre din clasa **ios** sunt utile în special atunci când aveţi nevoie să citiţi sau să scrieţi buffere de date.

Streamuri de intrare/ieşire bazate pe matrice

Pentru a crea un stream bazat pe matrice care efectuează atât intrări cât şi ieşiri, folosiţi această funcţie constructor tip **strstream**:

strstream iostr(char **buffer*, int *marime*, int *mod*);

Aici, *buffer* indică spre un şir care va fi folosit pentru operaţii de I/O. Valoarea din *marime* specifică mărimea matricei. Valoarea din *mod* determină cum operează streamul. Pentru operaţii de I/O normale, aceasta va fi **ios::in | ios::out**. Pentru intrare, matricea va trebui să se încheie cu null.

Iată un program care foloseşte o matrice pentru a efectua atât intrări cât şi ieşiri:

```
// Efectueaza atat intrari cat si iesiri.
#include <iostream.h>
#include <strstrea .h>

main()
{
    char iostr[80];

    strstream ios(iostr, sizeof(iostr), ios::in | ios::out);

    int a, b;
    char sir[80];
```

```
    ios << "10 20 testare";
    ios >> a >> b >> sir;
    cout << a << " " << b << " " << sir << endl;

    return 0;
}
```

Pentru început, el scrie **10 20 testare** în **iostr** şi apoi citeşte această informaţie din **iostr**.

Accesul aleator în cadrul matricelor

Amintiţi-vă că tuturor operaţiilor de I/O bazate pe matrice li se aplică acţiunile de I/O normale, deci şi accesul aleator folosind **seekg()** şi **seekp()**. De exemplu, următorul program caută al optulea caracter în matricea **iostr** şi îl afişează. (El va afişa **h**.)

```
#include <iostream.h>
#include <strstrea .h>

main()
{
    char iostr[80];

    strstream ios(iostr, sizeof(iostr), ios::in |ios::out);

    char ch;
    ios << "abcdefghijklmnopqrstuvwxyz";
    ios.seekg(7, ios::beg);
    ios >> ch;
    cout << "Caracterul in pozitia 7: " << ch;

    return 0;
}
```

Puteţi să căutaţi oriunde *în interiorul* matricei de I/O; nu însă şi dincolo de limitele sale.

Streamurilor bazate pe matrice li se pot aplica, de asemenea, funcţii ca **tellg()** şi **tellp()**.

Utilizarea matricelor dinamice

În prima parte a acestui capitol, când aţi legat un stream de o matrice de ieşire, matricea şi mărimea sa au fost transmise constructorului **ostrstream**. Această

caracteristică este bună atât timp cât cunoaşteţi numărul maxim de caractere pe care îl veţi pune în acea matrice. Totuşi, ce se întâmplă dacă nu ştiţi cât de mare trebuie să fie matricea de ieşire? Soluţia problemei este să folosiţi a doua formă a constructorului **ostrstream**, prezentată aici:

ostrstream();

Când este folosit acest constructor, **ostrstream** creează şi întreţine o matrice alocată dinamic. Acestei matrice îi este permis să crească în lungime pentru a corespunde ieşirii pe care trebuie să o stocheze.

Reţineţi că funcţia constructor **ostrstream** *nu* returnează efectiv un pointer către matricea alocată. Pentru a avea acces dinamic la o astfel de matrice, trebuie să folosiţi o a doua funcţie numită **str()**. Ea „îngheaţă" matricea şi returnează un pointer către ea. O dată ce o matrice dinamică este îngheţată, nu mai poate fi folosită pentru o nouă ieşire. De aceea, nu îngheţaţi matricea înainte să terminaţi de transmis caractere.

Iată un program care foloseşte o matrice de ieşire dinamică:

```
#include <iostream.h>
#include <strstrea .h>

main()
{
    char *p;

    ostrstream outs; // matrice alocata dinamic

    outs << "Imi place C++ " ";
    outs << -10 << hex << " ";
    outs.setf(ios::showbase);
    outs << 100 << ends;

    p = outs.str(); /* Ingheata bufferul dinamic si
                       returneaza un pointer spre el. */
    cout << p;

    delete p; // Elibereaza bufferul dinamic creat de
              // ostrstream().
    return 0;
}
```

Aşa cum ilustrează acest program, o dată ce matricea dinamică a fost îngheţată, vă revine responsabilitatea să eliberaţi memoria pentru sistem când aţi

terminat cu ea. Dacă nu ați înghețat deloc matricea, memoria este eliberată automat când este distrus streamul.

Puteți, de asemenea, să folosiți matricele dinamice de I/O cu clasa **strstream**, care poate să efectueze atât intrări cât și ieșiri cu o matrice. Pentru a crea o matrice dinamică folosind **strstream**, folosiți această construcție:

strstream();

Aceasta va crea o matrice dinamică ce va fi capabilă să creeze intrări și ieșiri.

Manipulatori și operații de I/O bazate pe matrice

Deoarece streamurile bazate pe matrice sunt aceleași ca și oricare alt stream, manipulatorii pe care îi creați în general pentru I/O pot fi folosiți fără nici o modificare și de operațiile de I/O bazate pe matrice. De exemplu, în **Capitolul 17**, au fost creați manipulatorii de ieșire **sd()** și **ss()** (săgeată la dreapta și, respectiv, săgeată la stânga) pentru I/O de la consolă. Următorul program arată că ei sunt la fel de eficienți și pentru operații de I/O bazate pe matrice.

```
// Acest program foloseste manipulatorii creati de
// utilizator pentru operatii de I/O bazate pe matrice
#include <strstrea .h>
#include <iostream.h>

// Sageata la dreapta
ostream &sd(ostream &stream)
{
    stream << "-------> ";
    return stream;
}

// Sageata la stanga
ostream &ss(ostream &stream)
{
    stream << " <-------";
    return stream;
}

main()
{
    char sir[80];

    ostream outs(sir, sizeof(sir));
    outs << sd << "Priviti acest numar: ";
    outs << 1000000 << ss << ends; // terminat cu null

    cout << " " << sir;

    return 0;
}
```

Acest program afișează următoarea ieșire:

```
-------> Priviti acest numar: 1000000 <-------
```


Funcții create de utilizator pentru extragere și inserție

Deoarece streamurile bazate pe matrice sunt ca oricare alte streamuri, puteți să creați propriile funcții de extragere și de inserție, în același fel în care le creați pentru alte tipuri de streamuri. De exemplu, următorul program construiește o clasă numită **punct**, care memorează coordonatele x, y ale unui punct în spațiul bidimensional. Funcția de inserție supraîncărcată pentru această clasă afișează un mic plan de coordonate și indică poziția punctului. Pentru simplificare, mărimea coordonatelor x, y este restrânsă la intervalul dintre 0 și 5.

```
#include <iostream.h>
#include <strstrea .h>

const int marime = 5;

class punct {
    int x, y;
public:
    punct(int i, int j) {
    // pentru acest exemplu se limiteaza x si y la
    // intervalul dintre 0 si marime
        if(i>marime) i = marime; if (i<0) i=0;
        if(j>marime) j = marime; if (j<0) j=0;
        x=i; y=j;
    }

    // O functie de insertie pentru punct.
```

```
    friend ostream &operator<<(ostream &stream, punct O);
};

ostream &operator<<(ostream &stream, punct O)
{
    register int i, j;

    for(j=marime; j>=0; j--) {
        stream << j;
        if(j == o.y) {
            for(i=0; i<O.x; i++) stream << " ";
            stream << '*';
        }
        stream << "\n";
    }

    for(i=0; i<=marime; i++) stream << " " << i;
    stream << "\n";

    return stream;
}

main()
{
    punct a(2, 3), b(1, 1);
    char sir[200];
    // afiseaza folosind mai intai cout
    cout << "Afiseaza folosind cout:\n";
    cout << a << "\n" << b << "\n\n";
    // acum foloseste operatii de I/O bazate pe RAM
    ostream outs(sir, sizeof(sir));

    // acum afiseaza outs folosind si formatarea in RAM
    outs << a << b << ends;

    cout << "Afiseaza folosind formatarea in RAM:\n";
    cout << sir;

    return 0;
}
```

Acest program determină următoarea ieşire:


```
Afiseaza folosind cout:
5
4
3  *
2
1
0
 0 1 2 3 4 5
5
4
3
2
1 *
0
 0 1 2 3 4 5

Afiseaza folosind formatarea in RAM:
5
4
3  *
2
1
0
 0 1 2 3 4 5
5
4
3
2
1 *
0
 0 1 2 3 4 5
```

Utilizări ale formatării bazate pe matrice

În C, funcţiile **sprintf()** şi **sscanf()** sunt folosite în special pentru pregătirea ieşirilor către sau citirii intrărilor de la echipamente nestandardizate. Însă, datorită capacităţilor lui C++ de a supraîncărca funcţiile de inserţie şi de extragere relativ la clase şi de a crea manipulatori adaptaţi, pot fi manevrate uşor multe echipamente exotice. Aceasta face ca necesitatea formatării în RAM să fie mai puţin importantă. Totuşi, există încă multe utilizări pentru operaţii de I/O bazate pe matrice.

O utilizare curentă a unei formatări bazate pe matrice este să construiţi un şir care să fie folosit ca intrare ori pentru o bibliotecă standard ori pentru o terţă

funcţie. De exemplu, puteţi să aveţi nevoie să construiţi un şir care să fie analizat de funcţia de bibliotecă standard **strtok()**. (Funcţia **strok()** „analizează" - adică descompune în elemente - un şir.) Un alt loc în care pot fi utilizate operaţiile de I/O bazate pe matrice este în editoarele de texte care efectuează operaţii complexe de formatare. Deseori, este mai uşor să folosiţi operaţiile de I/O din C++ bazate pe matrice pentru a construi un şir complex, decât s-o faceţi manual. Foarte utilă în programarea în Windows este şi construirea şirurilor folosind formatarea bazată pe matrice. Windows nu conţine nici o funcţie standard care să asigure ieşiri formatate într-o fereastră, ci trebuie să construiţi din timp toate ieşirile formatate.

Capitolul 20

Şabloane



O caracteristică relativ nouă în C++ este *şablonul* (template). Cu un şablon este posibil să creaţi *funcţii generice* şi *clase generice*. Într-o funcţie sau clasă generică, tipul de date asupra căruia operează acestea este specificat ca un parametru. De aceea, puteţi folosi o funcţie sau o clasă cu mai multe tipuri de date diferite, fără să rescrieţi versiunile specifice acestora. Aici sunt discutate atât funcţiile cât şi clasele generice.

NOTĂ: *Şabloanele nu au făcut parte dintre specificaţiile originale ale limbajului C++, dar au fost adăugate în 1990. Ele sunt definite de către standardul ANSI C++ propus şi sunt admise de majoritatea compilatoarelor de C++ disponibile astăzi.*

Funcţii generice

O funcţie generică defineşte un set general de operaţii care vor fi aplicate unor tipuri de date variate. Unei astfel de funcţii tipul de date asupra căruia va opera îi este transmis ca parametru. Utilizând acest mecanism, poate fi aplicată aceeaşi procedură unui domeniu larg de date. După cum probabil ştiţi, mulţi algoritmi au aceeaşi logică, indiferent de tipul de date asupra căruia operează. De exemplu, algoritmul de sortare Quicksort este acelaşi chiar dacă se aplică unei matrice de întregi sau uneia de tip **float**. Ceea ce diferă este doar tipul de date care este sortat. Creând o funcţie generică, puteţi defini natura algoritmului, independent de date. O dată făcut acest lucru, atunci când se execută funcţia, compilatorul generează automat codul corect pentru tipul de date folosit efectiv. În esenţă, când creaţi o funcţie generică, creaţi o funcţie care se supraîncarcă singură, automat.

O astfel de funcţie este creată cu ajutorul cuvântului cheie **template**. Semnificaţia normală a cuvântului „şablon" reflectă exact utilizarea sa în C++. El este folosit pentru a crea un şablon (tipar, model) care descrie ce va face o funcţie, lăsând compilatorul să completeze detaliile necesare. Iată forma generală a unei definiţii de funcţie de tip **template**:

template <class *Tip*> *tip-retur nume-func(lista parametri)*
{
    // corpul funcţiei
}

Tip este un nume care ţine locul tipului de date folosite de către funcţie. Acest nume poate fi folosit în cadrul definirii unei funcţii. Dar, el ţine doar un loc pe care compilatorul îl va înlocui automat cu tipul de date efectiv, atunci când va crea o versiune specifică a funcţiei.

Următorul exemplu scurt creează o funcţie generică ce inversează între ele valorile celor două variabile cu care este apelată. Deoarece procesul general de

înlocuire a două valori este independent de tipul acestora, este foarte bine să îl descrieţi într-o funcţie generică.

```
// Exemplu de functie sablon.
#include <iostream.h>
// Aceasta este o functie sablon.
template <class X> void inloc(X  &a, X &b)
{
    X temp;

    temp = a;
    a = b;
    b = temp;
}

main()
{
    int i=10, j=20;
    float x=10.1, y=23.3;
    char a='x', b='z';

    cout << "i, j originali: " << i << ' ' << j << endl;
    cout << "x, y originali: " << x << ' ' << y << endl;
    cout << "a, b originali: " << a << ' ' << b << endl;
    inloc(i,j); // inverseaza intregii
    inloc(x, y); // inverseaza float
    inloc(a,b); // inverseaza caractere
    cout << "i, j inversati: " << i << ' ' << j << endl;
    cout << "x, y inversati: " << x << ' ' << y << endl;
    cout << "a, b inversati: " << a << ' ' << b << endl;
    return 0;
}
```

Să privim cu atenţie acest program. Linia:

```
template <class X> void inloc(X &a, X &b)
```

spune compilatorului două lucruri: că este creat un şablon şi că va urma definirea generică. **X** este aici un tip generic care este folosit ca substitut. După zona **template** este declarată funcţia **inloc()**, folosind pe **X** ca tip de date pentru valorile care vor fi inversat. În **main()**, funcţia **inloc()** este apelată cu trei tipuri de date: **int**, **float** şi **char**. Deoarece **inloc()** este o funcţie generică, compilatorul îi va crea automat trei versiuni - una care va inversa valorile întregi, una care va inversa pe cele în virgulă mobilă şi una care va inversa caractere.

Iată câţiva alţi termeni care sunt folosiţi uneori atunci când vine vorba despre şabloane şi pe care îi puteţi întâlni în literatura de C++. Mai întâi, o funcţie generică (adică o definire a unei funcţii precedate de declararea **template**) este numită şi *funcţie şablon*. Când compilatorul creează o versiune concretă a acestei funcţii se spune că a creat o *funcţie generată*. Procesul de generare a unei funcţii este numit *de exemplificare* (*de instanţiere*). Altfel spus, o funcţie generată este un exemplar specific al unei funcţii şablon.

Practic, zona **template** a definirii unei funcţii generice nu trebuie să fie pe aceeaşi linie cu numele funcţiei. De exemplu, următorul fragment este, de asemenea, un mod uzual de a scrie funcţia **inloc()**.

```
template <class X>
void inloc(X &a, X &b)
{
    X temp;

    temp = a;
    a = b;
    b = temp;
}
```

Dacă folosiţi această formă, este important să înţelegeţi că între instrucţiunea **template** şi începutul funcţiei generice nu poate să apară nici o altă instrucţiune. De exemplu, fragmentul prezentat mai jos nu va fi compilat.

```
// Acesta nu va fi compilat
template <class X>
int i; // aceasta este greseala
void inloc(X &a, X &b)
{
    X temp;

    temp = a;
    a = b;
    b = temp;
}
```

După cum arată comentariul, specificarea **template** trebuie să fie imediat urmată de definiţia funcţiei.

O funcţie cu două tipuri generice

Într-o instrucţiune **template** puteţi să definiţi mai mult de un tip generic, folosind o listă separată prin virgulă. De exemplu, următorul program creează o funcţie generică cu două tipuri generice.


```
#include <iostream.h>

template <class tip1, class tip2>
void funcmea(tip1 x, tip2 y)
{
    cout << x << ' ' << y << endl;
}
main()
{
    funcmea(10, "hi");

    funcmea(0.23, 12L);

    return 0;
}
```

În acest exemplu, substituenţii **tip1** şi **tip2** sunt înlocuiţi de către compilator cu tipurile de date **int** şi **char***, respectiv **double** şi **long**, atunci când generează exemplarele specifice pentru **funcmea()** din cadrul funcţiei **main()**.



REŢINEŢI: *Când creaţi o funcţie generică, permiteţi, de fapt, compilatorului să genereze atâtea versiuni ale funcţiei câte sunt necesare pentru a trata modurile diferite în care funcţia este apelată de către programul dvs.*

Supraîncărcarea explicită a unei funcţii generice

Chiar dacă o funcţie şablon se supraîncarcă singură când este necesar, puteţi să o supraîncărcaţi şi explicit. Dacă supraîncărcaţi o funcţie generică, atunci ea suprascrie (sau „ascunde") funcţia generică relativ la acea versiune specifică. Să luăm, de exemplu, această variantă a primului exemplu:

```
// Suprascrierea unei functii sablon.
#include <iostream.h>

template <class X> void inloc(X  &a, X &b)
{
    X temp;

    temp = a;
    a = b;
```

```
    b = temp;
}

// aceasta suprascrie versiunea generica a inloc().
void inloc(int &a, int &b)
{
    int temp;

    temp = a;
    a = b;
    b = temp;
    cout << "In functia inloc(int &, int &) suprascrisa.\n";
}

main()
{
    int i=10, j=20;
    float x=10.1, y=23.3;
    char a='x', b='z';

    cout << "i, j originali: " << i << ' ' << j << endl;
    cout << "x, y originali: " << x << ' ' << y << endl;
    cout << "a, b originali: " << a << ' ' << b << endl;

    inloc(i,j); // aceasta apeleaza inloc()supraincarcat
                // explicit
    inloc(x, y); // inverseaza float
    inloc(a,b); // inverseaza caractere
    cout << "i, j inversate: " << i << ' ' << j << endl;
    cout << "x, y inversate: " << x << ' ' << y << endl;
    cout << "a, b inversate: " << a << ' ' << b << endl;
    return 0;
}
```

După cum menționează comentariile, la apelarea funcției **inloc(i, j)** este invocată versiunea sa redefinită explicit în program. Astfel, compilatorul nu generează această variantă a funcției, deoarece funcția generică pierde prioritatea în fața celei supraîncărcate explicit.

Supraîncărcarea manuală a șablonului, așa cum s-a arătat în acest exemplu, vă permite să scrieți o versiune a funcției generice adaptată la o situație specială. Totuși, în general, dacă aveți nevoie de mai multe versiuni ale unei funcții pentru diferite tipuri de date, ar trebui să folosiți, mai degrabă, șablonul și nu supraîncărcarea.

Restricții pentru funcția generică

Funcțiile generice sunt similare celor supraîncărcate, doar că ele sunt mai restrictive. Când este supraîncărcată o funcție, în interiorul fiecărei variante se pot efectua acțiuni diferite. Dar o funcție generică trebuie să efectueze aceeași acțiune pentru toate versiunile - doar tipul de date poate să difere. De exemplu, în programul următor, funcțiile supraîncărcate nu pot fi înlocuite de funcții generice deoarece ele nu efectuează aceleași lucruri.

```
#include <iostream.h>
#include <math.h>

void funcmea(int i)
{
    cout << "valoarea este: " << i << \n";
}

void funcmea(double d)
{
    double parteintrg;
    double partefrac;

    partefrac = modf(d, &parteintrg);
    cout << "Partea fractionara: " << partefrac;
    cout << "\n";
    cout << "Partea intreaga: " << partintrg;
}

main()
{
    funcmea(1);
    funcmea(12.2);

    return 0;
}
```

Iată alte câteva restricții ale funcțiilor șablon. O funcție virtuală nu poate fi funcție șablon. Destructorii nu pot fi șabloane. O funcție șablon trebuie să folosească editoarele de legături din C++. (Aceasta înseamnă că ea nu poate folosi o specificare de editare de legături. Vedeți **Capitolul 22.**)

Aplicarea funcţiilor generice

Funcţiile generice reprezintă una dintre cele mai folositoare caracteristici ale limbajuluilui C++. Ele pot fi aplicate oricăror situaţii. Cum s-a menţionat mai devreme, oricând aveţi o funcţie care defineşte un algoritm general, puteţi să-l introduceţi într-o funcţie generică. O dată ce aţi făcut aşa ceva, puteţi să o folosiţi cu orice tip de date, fără să mai fie necesar să o rescrieţi. Înainte de a continua cu clasele generice, vor fi date două exemple de aplicare a funcţiilor generice. Ele ilustrează cât de uşor este să profitaţi de această facilitate puternică din C++.

O sortare generică

Sortarea este tipul de operaţie pentru care sunt destinate funcţiile generice. Într-o mare măsură un algoritm de sortare este acelaşi, indiferent de tipul de date care se sortează. Următorul program ilustrează aceasta, creând o sortare generică prin amestecare. Deşi sortarea prin amestecare este un algoritm puţin eficient, modul său de operare este clar şi evident şi oferă un exemplu uşor de înţeles. (Poate că veţi dori să încercaţi să creaţi o versiune generică pentru algoritmul dvs. de sortare preferat.) Funcţia **amestec()** va sorta orice tip de matrice. Ea este apelată cu un pointer către primul ei element şi cu numărul ei de elemete.

```
// Un caz de sortare prin amestecare.
#include <iostream.h>
template <class X> void amestec(
   X *elemente, // pointer spre matricea de sortat
   int numar) // numarul de elemente din matrice
{
   register int a, b;
   X t;

   for(a=1; a<numar; a++)
         for(b=numar-1; b>=a; b--)
               if(elemente[b-1] > elemente[b]) {
                  // inverseaza elementele
                  t = elemente[b-1];
                  elemente[b-1] = elemente[b];
                  elemente[b] = t;
               }
   }

main()
{
   int imatrice[7] = {7, 5, 4, 3, 9, 8, 6};
   double dmatrice[5] = {4.3, 2.5, -0.9, 100.2, 3.0};

   int i;

   count << "Iata matricea de intregi nesortata: ";
   for(i=0; i<7; i++)
         cout << imatrice[i] << ' ';
   cout << endl;
   cout << "Iata matricea de double nesortata: ";

   for(i=0; i<5; i++)
         cout << dmatrice[i] << ' ';
   cout << endl;

   amestec(imatrice, 7);
   amestec(dmatrice, 5);

   cout << "Iata matricea de intregi sortata: ";
   for(i=0; i<7; i++)
         cout << imatrice[i]; << ' ';
   cout << endl;
   cout << "Iata matricea de double sortata: ";
   for(i=0; i<5; i++)
         cout << dmatrice[i] << ' ';
   cout << endl;

   return 0;
}
```


Acest program generează următoarea ieşire:

```
Iata matricea de intregi nesortata: 7 5 4 3 9 8 6
Iata matricea de double nesortata: 4.3 2.5 -0.9 100.2 3
Iata matricea de intregi sortata: 3 4 5 6 7 8 9
Iata matricea de double sortata: -0.9 2.5 3 4.3 100.2
```

După cum puteţi vedea, programul precedent creează două matrice: una pentru întregi şi una pentru **double**. Apoi le sortează pe fiecare. Deoarece **amestec()** este o funcţie şablon, ea este supraîncărcată automat pentru a se adapta celor două tipuri de date diferite.

Compactarea unei matrice

O altă funcţie care beneficiază de crearea sa ca funcţie şablon este numită **compact()**. Ea compactează elementele unei matrice. După cum ştiţi probabil din

experienţa de programare, este uzual să doriţi să ştergeţi nişte elemente de undeva dintr-o matrice şi apoi să deplasaţi celelalte elemente, astfel încât spaţiile nefolosite să ajungă la sfârşit. Acest fel de operaţie este acelaşi pentru toate tipurile de matrice deoarece el este independent de tipul de date asupra căruia lucrează efectiv. Funcţia generică **compact()** prezentată în programul următor este apelată cu un pointer spre primul element al matricei, numărul de elemente din aceasta şi cu primul şi ultimul indice al elementelor care trebuie extrase. Apoi funcţia le şterge şi compactează matricea. În scop demonstrativ, ea pune zero în locul elementelor nefolosite de la sfârşitul matricei care au fost eliberate prin compactare.

```
// O functie generic de compactare a unei matrice.
#include <iostream.h>

template <class X> void compact(
    X *elemente, // pointer spre matricea ce trebuie compactata
    int numar, // numar de elemente din matrice
    int start, // indicele de pornire pentru zona compactata
    int stop) // indicele de incheiere a zonei compactate
{
    register int i;

    for(i=stop+1; i<numar; i++, start++)
         elemente[start] = elemente[i];
    /* Pentru demonstratie, restul matricei va fi
       completata cu 0. */
    for( ; start<numar; start++) element[start] = (X) 0;
}

main()
{
    int num[7] = {0, 1, 2, 3, 4, 5, 6};
    char sir[18] = "Functii generice";

    int i;

    cout << "Iata matricea de intregi, necompactata: ";
    for(i=0; i<7; i++)
         cout << num[i] << ' ';
    cout << endl;

    cout << "Iata sirul necompactat: ";
    for(i=0; i<17; i++)
         cout << sir[i] << ' ';
    cout << endl;

    compact(num, 7, 2, 4);
    compact(sir, 17, 6, 10);
    cout << "Iata matricea de intregi compactata: ";
    for(i=0; i<7; i++)
         cout << num[i] << ' ';
    cout << endl;

    cout << "Iata sirul compactat: ";
    for(i=0; i<17; i++)
         cout << sir[i] << ' ';
    cout << endl;
    return 0;
}
```


Acest program compactează două tipuri de matrice. Una este o matrice de întregi, iar cealaltă este un şir. Dar, funcţia **compact()** va avea efect pentru oricare tip de matrice. Iată ieşirea acestui program:

```
Iata matricea de intregi necompactata: 0 1 2 3 4 5 6
Iata sirul necompactat: F u n c t i i g e n e r i c e
Iata matricea de intregi compactata: 0 1 5 6 0 0 0
Iata sirul compactat: F u n c t i e r i c e
```

După cum ilustrează exemplul precedent, o dată ce începeţi să gândiţi în termeni de şabloane, vă vor veni spontan în minte multe utilizări. Atâta vreme cât logica de bază a unei funcţii este independentă de date, ea poate fi creată ca funcţie generică.

Clase generice

În afară de funcţiile generice, puteţi să definiţi, de asemenea, o clasă generică. Când faceţi asta, creaţi o clasă care defineşte toţi algoritmii folosiţi de ea, dar tipul de date care este manevrat efectiv va fi specificat ca un parametru la crearea obiectelor acelei clase.

Clasele generice sunt folositoare când o clasă conţine caracteristici generale. De exemplu, acelaşi algoritm care tratează o înşiruire de întregi va lucra şi pentru o înşiruire de caractere. De asemenea, acelaşi mecanism care întreţine liste înlănţuite pentru adrese poştale va gestiona şi liste pentru articole auto. Folosind o clasă generică, puteţi să definiţi operaţiile care vor controla pentru orice tip de date

înşiruirea, listele înlănţuite ş.a.m.d. Compilatorul va genera automat tipul corect al obiectului, bazat pe tipul pe care îl specificaţi atunci când este creat acesta.

Iată forma generală a declarării unei clase generice:

template<class *Tip*>class *nume-clasa* {
.
.
.
}

Aici, *Tip* ţine locul numelui tipului pe care îl veţi specifica când se defineşte un exemplar al clasei. Dacă este necesar, puteţi să definiţi mai mult de un tip de date generice, folosind o listă separată prin virgule.

O dată ce aţi construit o clasă generică, puteţi crea un anumit exemplar al acesteia, folosind forma generală:

nume-clasa<tip>ob;

Aici, *tip* este numele tipului de date cu care va opera clasa. Funcţiile membre ale claselor generice sunt automat şi ele însele generice.

În programul următor, clasa **stiva** (folosită prima dată în **Capitolul 11)** este rescrisă ca o clasă generică. Astfel, ea poate fi folosită pentru a asigura o memorie stivă pentru orice tip de obiect. În exemplul prezentat aici sunt create o stivă de caractere, una de întregi şi una de numere în virgulă mobilă.

```
// Prezinta o clasa generica pentru memoria stiva.
#include <iostream.h>

const int SIZE = 100;

// Aceasta creeaza o clasa generica pentru stiva.
template <class STip> class stiva {
    STip stv[SIZE];
    int vis;
public:
    stiva();
    ~stiva();
    void pune(STip i);
    STip scoate();
};

// functia constructor pentru stiva
template <class STip > stiva<STip>::stiva()
```



```
{
    vis = 0;
    cout << "Stiva este initializata\n";
}
/* Functia destructor a stivei
   Aceasta functie nu este necesara. Ea este inclusa
   doar pentru demonstratie. */
template <class STip> stiva<STip>::~stiva()
{
    cout << "Stiva este distrusa\n";
}

// Pune un obiect in stiva.
template <class STip> void stiva<STip>::pune(STip i)
{
    if(vis==SIZE) {
          cout << "Stiva este plina.";
          return;
    }
    stv[vis] = i;
    vis++;
}

// Scoate un obiect din stiva.
template <class STip> STip stiva<STip>::scoate()
{
    if(vis==0) {
          cout << "Stiva este vida.";
          return 0;
    }
    vis--
    return stv[vis];
    }

main()
{
    stiva<int> a; // creeaza stiva de intregi
    stiva<double> b; // creeaza stiva pentru double
    stiva<char> c; // creeaza stiva pentru caractere

    int i;

    // foloseste stivele pentru intregi si pentru double
```

```
    a.pune(1);
    b.pune(99.3);
    a.pune(2);
    b.pune(-12.23);

    cout << a.scoate() << " ";
    cout << a.scoate() << " ";
    cout << b.scoate() << " ";
    cout << b.scoate() << "\n";

    // prezinta stiva pentru caractere
    for(i=0; i<10; i++) c.pune((char) 'A'+i);
    for(i=0; i<10; i++) cout << c.scoate();
    cout << "\n";

    return 0;
}
```

După cum puteți vedea, declararea clasei generice este similară cu cea a unei funcții generice. Tipul de date generic este folosit în declarația clasei și a funcțiilor membre. Până când nu se declară nici un obiect al stivei nu se determină tipul de date efectiv. Când este declarat un anumit exemplar pentru **stiva**, compilatorul generează automat toate funcțiile și variabilele necesare tratării tipului de date efectiv. În acest exemplu, sunt declarate trei tipuri de stivă (una pentru întregi, una pentru **double** și una pentru caractere). Fiți atenți, în special, la aceste declarații:

```
stiva<int> a; // creeaza stiva de intregi
stiva<double> b; // creeaza stiva pentru double
stiva<char> c; // creeaza stiva pentru caractere
```

Observați cum este transmis tipul de date dorit în interiorul parantezelor unghiulare. Puteți să modificați tipul de date memorat de stivă, schimbând tipul de date specificate atunci când sunt create obiectele de tip **stiva**. De exemplu, folosind următoarea declarare, ați fi putut crea o altă stivă care să fi conținut pointeri pentru caractere.

```
stiva<char *> carpointstiv;
```

Puteți crea, de asemenea, stive pentru a memora tipurile de date pe care le creați. De exemplu, dacă doriți să memorați informații despre adrese, folosiți această structură:


```
struct adr {
    char nume[40];
    char strada[40];
    char oras[30];
    char judet[3];
    char zip[12];
}
```

Apoi, pentru a folosi **stiva** ca să generați o stivă care să memoreze obiecte de tip **adr**, folosiți o declarație ca aceasta:

```
stiva<adr> obiect;
```

După cum ilustrează clasa **stiva**, funcțiile și clasele generice asigură un instrument puternic pe care îl puteți folosi pentru a obține cât mai mult de la programele dvs. deoarece vă permit să definiți forma generală a unui obiect care poate fi folosit apoi cu oricare tip de date. Sunteți astfel salvați de plictiseala de a construi implementări separate pentru fiecare tip de date cu care doriți să lucreze clasa respectivă. Compilatorul creează automat, pentru dvs., versiunea specifică a clasei.

Un exemplu cu două tipuri de date generice

O clasă șablon poate avea mai mult decât un singur tip de date generic. Declarați pur și simplu toate tipurile de date necesare clasei într-o listă separată prin virgule, în cadrul specificației **template**. De exemplu, următorul program scurt creează o clasă care folosește două tipuri de date generice.

```
/* Acest exemplu foloseste doua tipuri de date generice
   intr-o definire de clasa.
*/
#include <iostream.h>

template <class Tip1, class Tip2> class clasamea
{
    Tip1 i;
    Tip2 j;
public:
    clasamea(Tip1 a, Tip2  b) {i = a; j = b; }
    void arata() { cout << i << ' ' << j << '\n'; }
};

main()
```

```
{
    clasamea<int, double> ob1(10, 0.23);
    clasa mea<char, char *> ob2('X', "Acesta este un test");

    ob1.arata(); // arata int, double
    ob2.arata(); // arata char, char *

    return 0;
}
```

Acest program are următoarea ieşire:

```
10 0.23
X Acesta este un test
```

Programul declară două tipuri de obiecte. **ob1** foloseşte întregi, iar **ob2** foloseşte caractere şi pointeri pentru caractere. În ambele cazuri compilatorul generează automat datele şi funcţiile corespunzătoare pentru a se adapta felului în care s-a creat obiectul.

Crearea unei clase generice de matrice

Să urmărim o aplicaţie obişnuită a unei clase generice. După cum aţi văzut în Capitolul 14, puteţi să supraîncărcaţi operatorul [ ]. Făcând asta, puteţi crea propriile implementări de matrice. Aceasta vă permite crearea de „matrice sigure", care vă asigură verificarea limitelor în timpul rulării. După cum ştiţi, este posibil ca în C++ să depăşiţi (sau să fiţi sub) limita unei matrice în timpul rulării, fără să vă apară un mesaj de eroare. În schimb, dacă stabiliţi o clasă care conţine matricea şi permite accesul la ea doar prin operatorul pentru indice [ ] supraîncărcat, atunci puteţi să depistaţi un indice în afara limitelor.

Combinând supraîncărcarea operatorului cu o clasă generică, este posibil să creaţi un tip de matrice generică sigură, ce poate fi folosită pentru a crea matrice sigure de orice tip de date, după cum se arată în următorul program.

```
// Un exemplu de matrice generica sigura.
#include <iostream.h>
#include "stdlib.h"

const int SIZE = 10;

template <class ATip> class atip {
   ATip a[SIZE];
public:
```



```
    atip() {
        register int i;
        for(i=0; i<SIZE; i++) a[i] = i;
    }
    ATip &operator[](int i);
};
// Asigura verificarea limitelor pentru atip.
template <class ATip> ATip &atip<ATip>::operator[](int i)
{
    if(i<0; || i> SIZE-1) {
        cout << "\nValoarea de ";
        cout << i << " a indicelui este in afara limitei.\n";
        exit(1);
    }

    return a[i];
}

main()
{
    atip<int> intob; // matrice de intregi
    atip<double> doubleob; // matrice de double

    int i;

    cout << "Matrice de intregi: ";
    for(i=0; i<SIZE; i++) intob[i] = i;
    for(i=0; i<SIZE; i++) cout << intob[i] << " ";
    cout << '\n';

    cout << "Matrice de double: ":
    cout.precision(2);
    for(i=0; i<SIZE; i++) doubleob[i] = (double) i/3;
    for(i=0; i<SIZE; i++) cout << doubleob[i] << " ";
    cout << "\n';

    intob[12] = 100; // genereaza eroare in timpul rularii

    return 0;
}
```

Acest program implementează un tip de matrice generică sigură şi apoi demonstrează utilizarea sa creând o matrice de întregi şi una de **double**. (Puteţi să

încercaţi să creaţi şi alte tipuri de matrice.) După cum se arată în acest exemplu, o parte din puterea claselor generice este aceea că ele vă permit să scrieţi codul o dată, să îl depanaţi şi apoi să îl aplicaţi oricărui tip de date fără să îl rescrieţi pentru fiecare aplicaţie.

Pentru simplificare, programul anterior foloseşte matrice cu mărime fixă. Puteţi, însă, să schimbaţi clasa **atip** astfel încât să poată fi declarate matrice de dimensiuni variabile. Pentru a realiza aceasta, specificaţi dimensiunile matricei ca parametru al funcţiei constructor **atip** şi alocaţi matricea dinamic.

NOTĂ: În Capitolul 25 veţi găsi un alt exemplu de clasă generică. Ea creează o listă dublu înlănţuită capabilă să memoreze orice tip de obiect.

Capitolul 21

Tratarea excepţiilor



Acest capitol discută modul de *tratare a excepţiilor* în C++. *Tratarea excepţiilor* vă permite să rezolvaţi într-un mod ordonat erorile din timpul rulării; cu ajutorul acestui mecanism din C++, atunci când apare o eroare în timpul rulării, programul dvs. poate să apeleze automat o rutină de rezolvare a acesteia. Principalul avantaj al tratării excepţiilor este acela că el automatizează mult din codul de remediere a erorilor care, mai înainte, trebuia scris „de mână" în programele mai mari.

NOTĂ: Tratarea excepţiilor nu a făcut parte din specificaţia originală de C++, ci s-a dezvoltat din 1984 până în 1989, iar astăzi este definită în standardul ANSI C++ propus şi este admisă de majoritatea compilatoarelor disponibile.

Bazele tratării excepţiilor

Tratarea excepţiilor în C++ este construită pe trei cuvinte-cheie: **try**, **catch** şi **throw**. În termeni generali, instrucţiunile din program pe care doriţi să le urmăriţi în căutarea excepţiilor sunt conţinute într-un bloc **try**. Dacă o excepţie (ceea ce înseamnă o eroare) apare în blocul **try**, ea este „lansată" (folosindu-se **throw**), după care este „prinsă" cu instrucţiunea **catch** şi prelucrată. Următoarea discuţie se bazează pe această descriere generală.

După cum am spus, orice instrucţiune care lansează o excepţie trebuie să fie executată dintr-un bloc **try**. (Şi funcţiile apelate din interiorul blocului **try** pot, de asemenea, să lanseze o excepţie.) Orice excepţie trebuie să fie captată de instrucţiunea **catch**, ce urmează imediat după instrucţiunea **try**, care va lansa excepţia. Iată forma generală pentru **try** şi **catch**:

```
try {
    // bloc try
}
catch(tip1 arg) {
    // bloc catch
}
catch (tip2 arg) {
    // bloc catch
}
catch (tip3 arg) {
    // bloc catch
}
.
.
.
catch (tipN arg) {
    // bloc catch
}
```


Blocul **try** trebuie să conţină acea secţiune a programului în care doriţi să căutaţi erorile. Ea se poate întinde de la câteva instrucţiuni dintr-o funcţie până la întregul corp al funcţiei **main()** dintr-un bloc **try** (ceea ce determină urmărirea efectivă a întregului program).

Când este lansată o excepţie, ea este captată de instrucţiunea **catch** corespunzătoare, care o prelucrează. Pot fi mai multe instrucţiuni **catch** asociate uneia **try**. Care dintre instrucţiunile **catch** este folosită, se determină prin tipul excepţiei. Cu alte cuvinte, dacă tipul de date specificat de o ramură **catch** corespunde cu cel al excepţiei, atunci se execută acea instrucţiune **catch** (iar toate celelalte sunt ignorate). Când este prinsă o excepţie, valoarea sa va fi memorată de *arg*. Pot fi captate orice tipuri de date, inclusiv clasele pe care le creaţi. Dacă nu se găseşte nici o excepţie (deci că nu apare nici o eroare în blocul **try**), atunci nu se va executa nici o instrucţiune **catch**.

Iată forma generală a instrucţiunii **throw**:

throw *excepţie*;

throw trebuie să fie executată ori chiar din interiorul blocului **try**, ori din interiorul oricărei funcţii apelate (direct sau indirect) din blocul **try**. *excepţie* este valoarea lansată.

Dacă lansaţi o excepţie pentru care nu există nici o instrucţiune tip **catch** aplicabilă, poate să apară o încheiere anormală a programului. Dacă aveţi un compilator ce respectă standardul ANSI C++ propus, atunci lansarea unei excepţii netratate determină apelarea funcţiei **terminate()**. Implicit, **terminate()** apelează **abort()** pentru a opri programul dvs., dar puteţi specifica, dacă doriţi, propriul modul de încheiere. Pentru detalii, va trebui să consultaţi specificaţiile bibliotecii compilatorului.

Iată un exemplu simplu care arată felul în care operează tratarea excepţiilor în C++.

```
// Un exemplu simplu pentru tratarea exceptiilor.
#include <iostream.h>

main()
{
    cout << "Start\n";

    try { // inceputul blocului try
          cout << "In interiorul blocului try\n";
          throw 100; // lanseaza o eroare
          cout << "Aceasta nu se va executa";
```

```
    }
    catch (int i) { // preia eroarea
        cout << "Am prins o exceptie -- valoarea este: ";
        cout << i << "\n";
    }

    cout << "End";

    return 0;
}
```

Acest program va afişa următoarea ieşire:

```
Start
In interiorul blocului try
Am gasit o exceptie -- valoarea este: 100
End
```

Urmăriţi cu atenţie programul precedent. După cum puteţi vedea, există un bloc **try** care conţine trei instrucţiuni şi o ramură **catch(int i)** care prelucrează excepţia **int**. În interiorul blocului **try** vor fi executate doar două din cele trei instrucţiuni: prima instrucţiune **cout** şi instrucţiunea **throw**. O dată ce a fost găsită o excepţie, controlul trece la expresia **catch**, iar blocul **try** se încheie. Aceasta *nu* înseamnă că este apelat **catch**, ci că s-a executat un transfer al programului în acest punct (stiva programului s-a modificat automat ca atare. Astfel, instrucţiunea **cout** ce urmează după **throw** nu se va executa niciodată.

De obicei, codul din cadrul unei instrucţiuni **catch** încearcă să remedieze eroarea printr-o anumită acţiune. Dacă eroarea poate fi corectată, execuţia va continua cu instrucţiunile ce urmează după **catch**. Dar, de multe ori, o eroare nu poate fi reparată, iar blocul **catch** va termina programul cu un apel la **exit()** sau la **abort()**.

După cum s-a menţionat, tipul excepţiei trebuie să corespundă celui specificat în instrucţiunea **catch**. De exemplu, în programul precedent, dacă schimbaţi tipul din instrucţiunea **catch** în **double**, atunci excepţia nu va mai fi prinsă şi va apărea o terminare anormală a programului. Iată această modificare:

```
// Acest exemplu nu va functiona.
#include <iostream.h>

main()
{
    cout << "Start\n";
    try { // inceputul blocului try
        cout << "In interiorul blocului try\n";
        throw 100; // lanseaza o eroare
        cout << "Aceasta nu se va executa";
    }
    catch (double i) { // Nu va lucra pentru o exceptie int
        cout << "Am gasit o exceptie -- valoarea este: ";
        cout << i << "\n";
    }

    cout << "End"

    return 0;
}
```


Acest program va produce următoarea ieşire, deoarece excepţia de tip **int** nu va fi prinsă de către instrucţiunea **catch(double i)**.

```
Start
In interiorul blocului try
Încheiere anormala a programului (Abnormal program termination)
```

O excepţie poate fi lansată dintr-o instrucţiune care este în exteriorul blocului **try** atât timp cât se află într-o funcţie care este apelată din interiorul blocului **try**. De exemplu, acesta este un program valid:

```
/* Lansarea unei exceptii dintr-o functie din exteriorul
   blocului try
*/
#include <iostream.h>

void Xtest (int test)
{
    cout << "In interiorul lui Xtest, testul este: "
         << test << "\n";
    if(test) throw test;
}

main()
{
    cout << "Start\n";
    try { // inceputul blocului try
        cout << "In interiorul blocului try\n";
        Xtest(0);
```

```
        Xtest(1);
        Xtest(2);
    }
    catch (int i) { // preia o eroare
        cout << "Am gasit o exceptie -- valoarea este: ";
        cout << i << "\n";
    }

    cout << "Sfarsit";

    return 0;
}
```

Acest program va determina următoarea ieşire:

```
Start
In interiorul blocului try
In interiorul lui Xtest, testul este: 0
In interiorul lui Xtest, testul este: 1
Am gasit o exceptie -- valoarea este: 1
Sfirsit
```

Un bloc **try** poate fi localizat într-o funcţie. Când se întâmplă aşa ceva, mecanismul de tratare a erorilor referitor la funcţie este reiniţializat de câte ori este apelată aceasta. Să examinăm, de exemplu, acest program:

```
#include <iostream.h>
// Perechea try/catch poate exista si in alta functie decat
// main().
void Xmanip(int test)
{
    try{
        if(test) throw test;
    }
    catch(int i) {
        cout << "A gasit exceptia #:"<< i << '\n';
    }
}

main()
{
    cout << "Start\n";
```



```
    Xmanip(1);
    Xmanip(2);
    Xmanip(0);
    Xmanip(3);

    cout << "Sfarsit";

    return 0;
}
```

Acest program afişează următoarea ieşire:

```
Start
A gasit exceptia #: 1
A gasit exceptia #: 2
A gasit exceptia #: 3
Sfarsit
```

După cum puteţi vedea, sunt lansate trei excepţii. După fiecare excepţie, funcţia se returnează. Când aceasta este apelată din nou, se reiniţializează tratarea excepţiilor.

Este important de înţeles că un cod asociat cu instrucţiunea **catch** va fi executat doar dacă va capta o **excepţie**, iar altfel va fi ignorat. Execuţia nu va ajunge niciodată pe cale normală la instrucţiunea **catch**.) De exemplu, în următorul program nu este lansată nici o excepţie, astfel încât instrucţiunea **catch** nu se execută.

```
#include <iostream.h>

main()
{
    cout << "Start\n";
    try { // inceputul unui bloc try
        cout << "In interiorul blocului try\n";
        //    throw 100; // acest throw nu se executa
        cout << "Tot in blocul try\n";
    }
    catch (int i) { // gaseste o eroare
        cout << "A gasit o eroare -- valoarea este: ";
        cout << i << "\n";
    }

    cout << "Sfirsit";
```

```
    return 0;
}
```

Acest program determină următoarea ieşire:

```
Start
In interiorul blocului try
Tot in interiorul blocului try
Sfarsit
```

După cum puteţi vedea, instrucţiunea **catch** este ignorată de către execuţie.

Folosirea instrucţiunilor catch multiple

După cum am spus, unui **try** îi puteţi asocia mai mult de o instrucţiune **catch**. De fapt, chiar se obişnuieşte. Însă, fiecare **catch** trebuie să capteze un tip diferit de excepţie. De exemplu, acest program prinde atât întregi cât şi şiruri:

```
#include <iostream.h>

// Pot fi captate diferite tipuri de exceptii.
void Xmanip(int rest)
{
    try{
          if(test) throw test;
          else throw "Valoarea este zero";
    }
    catch(int i) {
          cout << "A preluat exceptia #: " << i << '\n';
    }
    catch(char *sir) {
          cout << "A preluat un sir: ";
          cout << sir << '\n';
    }
}

main()
{

    cout << "Start\n";
    Xmanip(1);
    Xmanip(2);
```



```
    Xmanip(0);
    Xmanip(3);

    cout << "Sfarsit";

    return 0;
}
```

Acest program determină următoarea ieşire:

```
Start
A preluat exceptia #: 1
A preluat exceptia #: 2
A preluat un sir: Valoarea este zero
A preluat exceptia #: 3
Sfarsit
```

După cum puteţi vedea, fiecare instrucţiune **catch** răspunde doar tipului său. Expresiile **catch** sunt verificate, în general, în ordinea în care apar în program. Se execută doar instrucţiunea care corespunde excepţiei, toate celelalte blocuri **catch** fiind ignorate.

Opţiuni de tratare a excepţiilor

În C++ există multe caracteristici şi nuanţe suplimentare pivind tratarea excepţiilor, care fac limbajul mai uşor şi mai convenabil de folosit. Ele sunt discutate în continuare.

Preluarea tuturor excepţiilor

În anumite circumstanţe veţi dori ca un manipulator al excepţiilor să le capteze pe toate, şi nu doar pe cele de un anumit tip. Aceast lucru este uşor de realizat. Folosiţi pur şi simplu această formă pentru **catch**:

```
catch(...) {
    // procesează toate excepţiile
}
```

Cele trei puncte corespund aici tuturor tipurilor de date. Următorul program ilustrează **catch(...)**.

```
// Acest exemplu captează toate exceptiile.
#include <iostream.h>

void Xmanip(int test)
{
    try{
          if(test==0) throw test; // lanseaza int
          if(test==1) throw 'a'; // lanseaza char
          if(test==2) throw 123.23; // lanseaza double
    }
    catch(...) { // preia toate exceptiile
          cout << "Am prins una!\n";
    }
}

main()
{
    cout << "Start\n";

    Xmanip(0);
    Xmanip(1);
    Xmanip(2);

    cout << "Sfarsit";

    return 0;
}
```

Acest program afişează următoarea ieşire:

```
Start
Am prins una!
Am prins una!
Am prins una!
Sfarsit
```

După cum puteţi vedea, toate cele trei **throw** au fost prinse printr-o singură instrucţiune **catch**.

O utilizare foarte bună pentru **catch(...)** este ca un ultim **catch** pentru o serie de **catch**. În această poziţie asigură o captare implicită utilă sau o instrucţiune de tip „prinde tot". De exemplu, această versiune uşor diferită a programului precedent preia explicit excepţii de tip întreg, dar se bazează pe **catch(...)** pentru a le capta pe celelalte.


```
// Acest exemplu foloseste catch(...) ca varianta implicita.
#include <iostream.h>

void Xmanip(int test)
{
    try{
          if(test==0) throw test; // lanseaza int
          if(test==1) throw 'a'; // lanseaza char
          if(test==2) throw 123.23; // lanseaza double
    }
    catch(int i) { // preia o exceptie int
          cout << "Am prins un intreg\n";
    catch(...) { // preia toate celelalte exceptii
          cout << "Am prins una!\n";
    }
}

main()
{
    cout << "Start\n";

    Xmanip(0);
    Xmanip(1);
    Xmanip(2);

    cout << "Sfarsit";

    return 0;
}
```

Iată ieşirea produsă de acest program:

```
Start
Am prins un intreg
Am prins una!
Am prins una!
Sfarsit
```

După cum sugerează exemplul, utilizarea instrucţiunii **catch(...)** implicite este o modalitate bună de a capta toate excepţiile pe care nu doriţi să le trataţi explicit. De asemenea, preluând toate excepţiile, evitaţi ca o excepţie netratată să determine o încheiere anormală a programului.

Restricții pentru excepții

Când o funcție este apelată dintr-un bloc **try**, puteți să restrângeți tipul de excepție pe care îl va lansa ea. De fapt, puteți preveni, de asemenea, ca funcția să lanseze orice fel de excepție. Pentru a realiza aceste restricții, trebuie să adăugați definirii funcției o clauză **throw**. Forma generală a acesteia este prezentată aici:

*tip-returnat nume-func(lista-arg)*throw*(lista-tip)*
{
    //...
}

Aici pot fi lansate de către funcție doar acele tipuri conținute în *lista-tip*, listă separată prin virgule. Lansarea oricăror alte tipuri de expresii va determina o încheiere anormală a programului. Dacă nu doriți ca o funcție să fie capabilă să elimine *nici* o expresie, atunci folosiți o listă goală.

Dacă aveți un compilator care respectă standardul ANSI C++ propus, atunci încercarea de a lansa o excepție care nu este admisă de către funcție va determina apelarea funcției **unexpected()**. Implicit, aceasta apelează funcția **abort()**, care duce la terminarea anormală a programului. Totuși, dacă doriți, puteți specifica propriul dvs. manipulator pentru încheiere. Pentru detalii va trebui să consultați biblioteca de referință a compilatorului.

Următorul program arată cum se restrâng tipurile de excepții ce pot fi lansate dintr-o funcție.

```
// Restrangerea tipurilor lansate de functie.
#include <iostream.h>

// Aceasta functie poate lansa doar int, char si double.
void Xmanip(int test) throw(int, char, double)
{
    if(test==0) throw test; // lanseaza test
    if(test==1) throw 'a'; // lanseaza char
    if(test==2) throw 123.23; // lanseaza double
}

main()
{
    cout << "Start\n";

try{
    Xmanip(0); // incercati, de asemenea, sa pasati 1 si 2
               // functiei Xmanip()
}
catch(int i) {
    cout << "Am prins un intreg\n";
}
catch(char c) {
    cout << "Am prins char\n";
}
catch(double d) {
    cout << "Am prins double\n";
}

cout << "Sfarsit";

return 0;
}
```


În acest program funcția **Xmanip()** poate să lanseze doar expresii de tip întreg, caracter și **double**. Dacă va încerca să lanseze orice alt tip de excepție, va produce o terminare anormală a programului. (Va fi apelată funcția **unexpected()**.) Pentru a vedea un asemenea exemplu, îndepărtați **int** din lista de argumente și rulați programul din nou.

Este important să înțelegeți că o funcție poate fi limitată doar în ceea ce privește tipurile de excepții pe care le lansează înapoi în blocul **try** care a apelat-o. Dar un bloc **try** *din interiorul* unei funcții poate să lanseze orice excepție, atât timp cât acesta este captat în *interiorul* funcției respective. Restricția se aplică doar când se lansează o excepție în afara funcției.

Următoarea modificare a funcției **Xmanip()** o împiedică să mai lanseze vreo excepție.

```
// Aceasta functie NU poate lansa nici o exceptie!
void Xmanip(int test) throw()
{
/* Urmatoarele instructiuni nu mai lucreaza,
   ele vor duce la o terminare anormala a programului. */
    if(test==0 throw test;
    if(test==1) throw 'a';
    if(test==2) throw 123.23;
}
```

Relansarea unei excepții

Dacă doriți să relansați o excepție din cadrul unui manipulator de excepții, puteți să o faceți apelând **throw** fără nici o excepție. Aceasta va determina ca excepția

curentă să fie transmisă unei secvenţe **try/catch** exterioare. Cel mai bun motiv pentru a face acest lucru este să permiteţi manipulatorilor multipli accesul la acele excepţii. De exemplu, poate că un manipulator de excepţii tratează un aspect al uneia, iar al doilea se ocupă de altul. O excepţie poate fi relansată doar din blocul **catch** (sau din orice funcţie apelată din interiorul acestuia). Când relansaţi o excepţie, ea nu va fi preluată de aceeaşi instrucţiune **catch**, ci se va deplasa spre următoarea. Programul de mai jos ilustrează relansarea unei excepţii, şi anume cea de tipul **char***.

```
// Exemplu de "relansare" a unei exceptii.
#include <iostream.h>

void Xmanip()
{
    try{
        throw "hello"; // elimina un char *
    }
    catch(char *) { // capteaza un char *
        cout << "Am prins un char * in interiorul
                functiei Xmanip\n";
        throw ; // relanseaza char * in afara functiei
    }
}

main()
{
    cout << "Start\n";

    try{
        Xmanip();
    }
    catch(char *) {
        cout << "Am prins un char * in interiorul
                functiei main\n";
    }

    cout << "Sfarsit";

    return 0;
}
```

Acest program afişează următoarea ieşire:


```
Start
Am prins un char * in interiorul lui Xmanip
Am prins un char * in interiorul lui main
Sfarsit
```

Aplicaţii ale tratării excepţiilor

Tratarea excepţiilor este destinată asigurării unei modalităţi structurate prin care programul dvs. să reacţioneze la evenimente neobişnuite. Aceasta înseamnă că, atunci când apare o eroare, manipulatorul să facă ceva raţional. De exemplu, să considerăm următorul program simplu, care introduce două numere şi împarte pe primul la al doilea. El foloseşte tratarea erorilor pentru a preveni o eroare de împărţire la zero.

```
#include <iostream.h>

void impart(double a, double b);

main()
{
    double i, j;
    do {
        cout << "Introduceti deimpartitul (0 pentru stop): ";
        cin >> i;
        cout << "Introduceti impartitorul: ";
        cin >> j;
        impart(i, j);
    }   while(i != 0);

    return 0;
}
void impart(double a, double b)
{
    try {
        if(!b) throw b; // verifica impartirea la 0
        cout << "Rezultat: " << a/b << endl;
    }
    catch (double b) {
        cout << "Nu pot imparti la zero.\n";
    }
}
```

Deşi programul precedent este foarte simplu, el ilustrează esenţa tratării excepţiilor. De vreme ce împărţirea la zero nu este permisă, programul nu poate continua dacă al doilea număr introdus este zero. În acest caz, excepţia este tratată prin neefectuarea împărţirii (care ar fi determinat o terminare anormală a programului) şi prin avertizarea utilizatorului asupra erorii. Programul solicită apoi din nou introducerea a două numere. Astfel, eroarea a fost tratată corect, iar utilizatorul poate să continue lucrul cu programul. Aceeaşi concepţie va sta şi la baza aplicaţiilor mai complexe pentru tratarea erorilor.

Tratarea erorilor este folositoare, în special, pentru a ieşi, atunci când apare o eroare catastrofală, dintr-un lanţ de rutine imbricate profund. În această privinţă, tratarea excepţiilor în C++ este proiectată pentru a înlocui funcţiile destul de greoaie din C, **setjump()** şi **longjump()**.



REŢINEŢI: Motivul principal pentru utilizarea tratării excepţiilor este de a asigura o modalitate ordonată de tratare a erorilor. Aceasta presupune, dacă este posibil, corectarea situaţiei.

Capitolul 22

Elemente diverse şi caracteristici avansate



Acest capitol prezintă câteva apecte din C++ care nu au fost examinate în altă parte a acestei carţi. Caracteristicile includ funcţiile de conversie, constructorii de copii, argumentele implicite ale funcţiilor, specificaţii pentru editarea legăturilor, şi alte elemente noi, adăugate de standardul propus pentru ANSI C++ sau diferenţe între C şi C++.

Argumente implicite pentru funcţii

C++ permite ca, atunci când nu se specifică un argument corespunzător pentru un parametru al unei funcţii, să i se atribuie acelui parametru o valoare implicită. Aceasta este specificată într-o manieră similară sintactic iniţializării unei variable.

Următorul exemplu declară **funcmea()** ca preluând un argument în virgulă mobilă, de tip **double**, cu valoarea implicită 0.0.

```
void funcmea(double d = 0.0)
{
    .
    .
    .
}
```

Acum, **funcmea()** poate fi apelată în două moduri, aşa cum arată următorul exemplu.

```
funcmea(198.234); // transmite explicit o valoare
funcmea(); // lasa functia sa foloseasca valoarea implicita
```

Prima apelare pasează lui **d** valoarea 198.234. Cea de-a doua îi dă automat valoarea implicită 0.

Un motiv pentru care argumentele implicite sunt incluse în C++ este acela că ele oferă programatorului o altă metodă pentru a trata programele de complexitate mai mare. Pentru a face faţă varietăţii mari de situaţii, o funcţie conţine destul de frecvent mai mulţi parametri decât sunt necesari pentru o utilizare normală. De aceea, când se aplică argumentele implicite, trebuie să vă amintiţi şi să specificaţi doar argumentele care sunt semnificative pentru cele mai uzuale situaţii, nu şi pentru cazul cel mai general. De exemplu, multe dintre funcţiile de I/O din C++ descrise în capitolele precedente folosesc argumente implicite exact din acest motiv.

O ilustrare simplă pentru cât de folositor poate fi un argument implicit al unei funcţii este realizată de către funcţia **stergecr()** din următorul program. Această funcţie şterge ecranul generând o serie de caractere de salt la linie nouă (nu este modul cel mai eficient, dar este suficient pentru acest exemplu). Deoarece un monitor obişnuit afişează 25 de linii de text, este furnizat argumentul implicit 25.

Dar, pentru că unele terminale pot să afişeze mai mult sau mai puţin de 25 de linii (de multe ori depinzând de modul video în care se lucrează), puteţi să supraîncărcaţi argumentul implicit specificând unul în mod explicit.

```
#include <iostream.h>

void stergecr(int marime=25);

main()
{
    register int i;

    for(i=0; i<30; i++) cout << i << endl;
    cin.get();
    stergecr(); // curata 25 de linii

    for(i=0; i<30; i++) cout << i << endl;
    cin.get()
    strergecr(10); // sterge 10 linii

    return 0;
}

void stergecr(int marime)
{
    for(; marime; marime--) cout << endl;
}
```

După cum ilustrează programul, nu este necesară specificarea nici unui argument la apelarea lui **stergecr()** atunci când valoarea implicită corespunde situaţiei. Este însă posibil ca, atunci când este necesar, să neglijaţi valoarea implicită şi să daţi o valoare diferită parametrului **marime**.

Un argument implicit poate fi folosit, de asemenea, ca indicator care spune funcţiei să reia un argument anterior. Pentru a ilustra acest tip de utilizare este prezentată o funcţie scurtă numită **intra()**, care indentează automat un şir cu un număr specificat. Iată, pentru început, o versiune a acestei funcţii care nu foloseşte un argument implicit:

```
void intra(char *sir, int aliniat)
{
    if(aliniat < 0) aliniat = 0;

    for( ; aliniat; aliniat--) cout << " ";
```

```
    cout << sir << "\n";
}
```

Această versiune a funcţiei **intra()** este apelată având ca prim argument şirul care va fi afişat şi ca al doilea argument mărimea indentării. Deşi forma funcţiei nu este cu nimic greşită, puteţi să îi măriţi utilitatea furnizându-i un argument implicit pentru parametrul **aliniat** care îi spune lui **intra()** să folosească indentarea specificată anterior. Este destul de uzuală afişarea unui bloc de text cu fiecare linie indentată la aceeaşi distanţă. În această situaţie, în loc să fie necesar să transmiteţi în mod repetat acelaşi argument pentru **aliniat**, puteţi să îi daţi o valoare implicită care îi spune lui **intra()** să indenteze ca mai înainte. Această abordare este ilustrată în programul următor:

```
#include <iostream.h>

/* Indentare implicita cu -1. Aceasta valoare ii spune
   functiei sa refoloseasca valoarea anterioara. */
void intra(char *sir, int aliniat = -1);

main()
{
    intra("Va salut", 10);
    intra("Acesta va fi indentat, implicit, cu 10 spatii");
    intra("Acesta va fi indentat cu 5 spatii", 5);
    intra("Acesta nu va fi indentat", 0;

    return 0;
}

void intra(char *sir, int aliniat)
{
    static i = 0; // pastreaza valoarea anterioara a
                  // aliniatului

    if(aliniat>=0)
          i = aliniat;
    else // refoloseste valoarea vechiului aliniat
          aliniat = i;

    for( ; aliniat; aliniat--) cout << " ";

    cout << sir << "\n";
}
```


Acest program afişează următoarea ieşire:

```
          Va salut
          Acesta va fi indentat, implicit, cu 10 spatii
     Acesta va fi indentat cu 5 spatii
Acesta nu va fi indentat
```

Când creaţi funcţii care au valori implicite pentru argumente, este important să vă amintiţi că aceste valori trebuie specificate doar o dată, şi anume prima dată când este declarată funcţia într-un fişier. În exemplul precedent argumentul implicit a fost menţionat în prototipul funcţiei **intra()**. Dacă veţi încerca să specificaţi o nouă valoare implicită (sau chiar aceeaşi) în definirea ei, compilatorul va afişa o eroare şi nu va compila programul. Deşi nu pot fi redefinite argumente implicite ale aceleiaşi funcţii, puteţi să specificaţi argumente implicite diferite pentru fiecare versiune a unei funcţii supraîncărcate.

Toţi parametrii care preiau valori implicite trebuie să apară la dreapta celor care nu preiau. De exemplu, este incorect să definiţi astfel **intra()**:

```
// gresit!
void intra(int aliniat = -1, char *sir);
```

O dată ce aţi definit parametrii care preiau valori implicite, nu puteţi să specificaţi un parametru care nu este implicit. Deci, o declaraţie ca aceasta este, de asemenea, greşită, şi nu va fi compilată:

```
int funcmea(float f, char *sir, int i=10, int j);
```

Deoarece lui **i** i s-a dat o valoare implicită, atunci şi lui **j** trebuie să i se dea una.

Puteţi folosi parametri impliciţi într-o funcţie constructor a unui obiect. De exemplu, clasa **cub** prezentată aici păstrează dimensiunile unui cub ca valori întregi. Funcţia sa constructor iniţializează implicit toate dimensiunile cu zero dacă nu se asigură nici un alt argument, aşa cum este prezentat mai jos:

```
#include <iostream.h>

class cub {
    int x, y, z;
public:
    cub(int i=0, int j=0, int k=0) {
          x=i;
          y=j;
          z=k;
    }
```

```
    int volum() {
        return x*y*z;
    }
};

main()
{
    cub a(2,3,4), b;

    cout << a.volum() << endl;
    cout << b.volum();

    return 0;
}
```

Includerea argumentelor implicite într-o funcţie constructor, atunci când este cazul, prezintă două avantaje. În primul rând, vă scuteşte să mai creaţi şi un constructor cu lista de parametri vidă. De exemplu, dacă parametrii funcţiei **cub()** nu ar fi fost impliciţi, ar fi fost nevoie şi de al doilea constructor, prezentat aici, care să trateze declararea lui **b** (care nu specifică nici un argument):

```
cub() {x=0; y=0; z=0}
```

În al doilea rând, introducerea valorilor iniţiale uzuale este mai convenabilă decât specificarea lor de fiecare dată când este declarat un obiect.

Utilizarea corectă a argumentelor implicite

Chiar dacă argumentele implicite pot fi un instrument puternic când sunt folosite corect, ele pot fi şi sursă de erori. Scopul principal al argumentelor implicite este să permită funcţiei să lucreze eficient, într-un mod uşor de utilizat, fără a-şi pierde însă din flexibilitate. Pentru aceasta, toate argumentele implicite ar trebui să se încadreze în varianta în care este folosită funcţia în majoritatea timpului. De exemplu, un argument implicit îşi are rostul dacă este folosit 90% din timp. Dar, dacă o valoare va fi întâlnită doar în 10% din apelări, iar în restul timpului argumentul care corespunde acestui parametru va varia mult, nu este recomandabil să stabiliţi un argument implicit. Rostul acestora este să furnizeze valorile pe care programatorul le asociază cu o anumită funcţie. Când nu există doar o singură valoare asociată curent unui parametru, nu există nici un motiv pentru crearea unui argument implicit. De altfel, declararea unor argumente implicite de care nu este realmente nevoie, va distruge structura codului creat de dvs. - va deruta şi va induce în eroare pe cei ce vor citi programul. Este, desigur, subiectiv cât să alegeţi între 10% şi 90% pentru ca argumentele implicite să devină cu adevărat utile, dar 51% pare o limită rezonabilă.

Un alt factor important pe care ar trebui să-l urmăriţi când folosiţi argumente implicite este ca nici un astfel de argument să nu determine o acţiune dăunătoare sau distructivă. Deci, utilizarea accidentală a unui argument implicit nu va determina o catastrofă.

Argumente implicite sau supraîncărcare?

Înainte de a părăsi subiectul argumentelor implicite, va fi discutată o altă aplicaţie. În unele situaţii, aceste argumente pot fi folosite ca o formă rapidă de supraîncărcare a funcţiei. Pentru a vedea de ce, imaginaţi-vă că doriţi să creaţi două versiuni proprii ale funcţiei standard **strcat()**. Prima versiune va lucra exact ca **strcat()** şi va adăuga întreg conţinutul unui şir la sfârşitul altuia. Cea de-a doua va prelua un al treilea argument care va specifica numărul de caractere de concatenare. A doua versiune va adăuga, deci, doar un anumit număr de caractere dintr-un şir la sfârşitul altuia. De aceea, presupunând că denumiţi funcţiile dvs. **strcatmeu()**, ele vor avea următoarele prototipuri:

```
void strcatmeu(char *s1, char *s2, int lung);
void strcatmeu(char *s1, char *s2);
```

Prima versiune va copia **lung** caractere din **s2** la sfârşitul lui **s1**. A doua versiune va copia întregul şir spre care indică **s2** la sfârşitul şirului spre care indică **s1** şi va lucra precum **strcat()**.

Deşi nu este greşit să implementaţi două versiuni ale funcţiei **strcatmeu()** pentru a realiza cele două efecte pe care le urmăriţi, există o cale mai simplă. Folosind un argument implicit, puteţi să creaţi doar o singură versiune a funcţiei **strcatmeu()** care să lucreze în ambele moduri. Următorul program prezintă acest procedeu.

```
// O versiune proprie pentru strcat().
#include <iostream.h>
#include <string.h>

void strcatmeu(char *s1, char *s2, int lung = 0);

main()
{
   char sir1[80] = "Acesta este un test";
   char sir2[80] = "0123456789";

   strcatmeu(sir1, sir2, 5); // concateneaza 5 caractere
```

```
    cout << sirl << 'n\';

    strcpy(sirl, "Acesta este un test"); // reinitializeaza sirl

    strcatmeu(sirl, sir2); // concateneaza intregul sir
    cout << sirl << '\n';

    return 0;
}
// O versiune proprie pentru strcat().
void strcatmeu(char *s1, char *s2, int lung)
{
    // gaseste sfarsitul sirului s1
    while(*s1) s1++;

    if(lung==0) lung = strlen(s2);

    while(*s2 && lung) {
        *s1 = *s2; // copiaza caractere
        s1++;
        s2++;
        lung--;
    }

    *s1 = '\0'; // terminare cu null pentru s1
}
```

Aici, **strcatmeu()** adaugă până la **lung** caractere din şirul spre care indică **s2** la sfârşitul şirului spre care indică **s1**. Însă, dacă **lung** este zero (ca în cazurile în care se păstrează valoarea implicită), **strcatmeu()** concatenează întregul şir spre care indică **s2** la sfârşitul lui **s1**. (De aceea, când **lung** este zero, funcţia operează ca şi funcţia **strcat()** standard.) Folosind argumentele implicite pentru **lung**, este posibil să combinaţi ambele operaţii într-o singură funcţie. În acest fel, argumentele implicite asigură uneori o formă rapidă de supraîncărcare a funcţiilor.

Crearea funcţiilor de conversie

În unele situaţii aveţi nevoie să folosiţi un obiect dintr-o clasă într-o expresie care implică alte tipuri de date. Uneori funcţiile operator supraîncărcate pot să asigure calea de a face aceasta. Dar, în alte cazuri, tot ceea ce doriţi este o simplă conversie a tipului clasei în tipul destinaţie. Pentru a trata acest caz, C++ vă permite să creaţi *funcţii proprii de conversie*. O funcţie de conversie transformă

tipul clasei într-un tip compatibil cu cel al restului expresiei. Forma generală a unei funcţii de conversie a tipului este:

operator *tip*() {return *valoare*;}

tip este aici tipul ţintă în care convertiţi clasa dvs., iar *valoare* este valoarea obiectului după conversie. Funcţiile de conversie returnează date de tipul *tip* şi nu este permis nici un alt specificator de tip pentru valoarea întoarsă. De asemenea, nu pot apărea parametri. O astfel de funcţie trebuie să fie un membru al clasei pentru care este definită. Funcţiile de conversie sunt moştenite şi pot fi virtuale.

Următoarea ilustrare a modului de creare a unei funcţii de conversie foloseşte clasa **stiva** prezentată în **Capitolul 11**. Să presupunem că doriţi să puteţi combina obiecte de tipul **stiva** cu o expresie de tip întreg. Mai mult, să presupunem că valoarea unui obiect din **stiva** folosită într-o expresie de tip întreg este numărul de valori aflate curent în memoria stivă. (Puteţi să doriţi să faceţi ceva de genul acesta dacă, de exemplu, folosiţi obiecte de tip **stiva** într-o simulare şi vreţi să urmăriţi cât de repede se umple stiva.) O cale de rezolvare este să convertiţi un obiect de tipul **stiva** într-un întreg care reprezintă numărul de elemente din memoria stivă. Pentru a realiza acest lucru, folosiţi o funcţie de conversie care arată astfel:

```
operator int() {return vis;}
```

Iată un program care ilustrează cum lucrează funcţiile de conversie:

```
#include <iostream.h>

const int SIZE=100;

// aceasta creeaza clasa stiva
class stiva {
    int stiv[SIZE];
    int vis;
public:
    stiva() {vis=0;}
    void pune(int i);
    int scoate(void);
    operator int() {return vis;} // conversia lui stiva in int
};

void stiva::pune(int i)
{
    if(vis==SIZE) {
```

```
            cout << "Stiva este plina.";
            return;
        }
        stiv[vis] = i;
        vis++;
}

int stiva::scoate()
{
    if(vis==0) {
            cout << "Stiva este vida.";
            return 0;
    }
    vis--;
    return stiv[vis];
}

main()
{
    stiva stiv;
    int i, j;

    for(i=0; i<20; i++) stiv.pune(i);

    j = stiv; // converteste in intreg

    cout << j << " elemente in stiva\n";

    cout << SIZE - stiv << " spatii libere\n";

    return 0;
}
```

Acest program afişează următoarea ieşire:

```
20 elemente in stiva
80 spatii libere
```

După cum ilustrează programul, când un obiect de tip **stiva** este folosit într-o expresie de tip întreg, aşa cum este **j = stiv**, obiectului i se aplică funcţia de conversie. În acest caz particular, funcţia returnează valoarea 20. De asemenea, funcţia este apelată când **stiv** este scăzut din **SIZE**.

Iată alt exemplu de funcţie de conversie. Acest program creează o clasă numită

putere(), care memorează şi calculează rezultatul unor numere ridicate la o putere. Ea obţine rezultatul în virgulă mobilă. Puteţi să folosiţi obiecte de tipul **putere()** în expresii care implică alte valori tip **double** asigurând o funcţie de conversie în tipul **double** şi returnând rezultatul.

```
#include <iostream.h>

class putere {
    double b;
    int e;
    double val;
public:
    putere(double baza, int exp);
    putere operator+(putere o) {
          double baza;
          int exp;
          baza = b +o.b;
          exp = e + o.e;

          putere temp(baza, exp);
          return temp;
    }
    operator double() {return val;} // conversie in double
};

putere::putere(double baza, int exp)
{
    b = baza;
    e = exp;
    val = 1;
    if(exp==0) return;
    for( ; exp>0; exp--) val = val * b;
}

main()
{
    putere x(4.0, 2);
    double a;

    a = x; // conversie in double
    cout << x + 100.2; // converteste x in double si aduna 100.2
    cout << "\n";
```

```
    putere y(3.3, 3), z(0, 0);

    z = x + y; // nici o conversie
    a = z; // conversie in double
    cout << a;

    return 0;
}
```

După cum puteți vedea, când **x** este utilizat în expresia **x + 100.2**, este folosită funcția de conversie pentru a determina o valoare de tip **double**. Rețineți, de asemenea, că în expresia **x + y** nu se aplică nici o conversie deoarece ea implică doar obiecte de tip **putere**.

După cum puteți deduce din exemplele anterioare, există multe situații în care este bine să creați o funcție de conversie pentru o clasă. Deseori aceste funcții asigură o sintaxă mai naturală când obiectele claselor sunt combinate cu tipuri încorporate. În particular, în cazul clasei **putere()**, posibilitatea conversiei în **double** face ca obiectele din acea clasă să fie „normale", mai uşor atât de programat cât şi de înțeles în expresiile matematice „normale".

Puteți crea funcții de conversie distincte pentru a preîntâmpina diverse cerințe. Ați putea defini una care converteşte, de exemplu, în **long**. Fiecare va fi aplicată automat în funcție de tipul expresiei.

Constructori pentru copii de obiecte

Implicit, când un obiect este folosit pentru a inițializa pe un altul, C++ efectuează o copie pe biți. Aceasta înseamnă că este creată o copie identică a obiectului inițial în obiectul țintă. Deşi metoda este adecvată perfect în multe cazuri - şi în general rezultatul este exact cel pe care îl doriți - există situații în care copia pe biți nu poate fi folosită. Una dintre cele mai obişnuite situații în care trebuie să evitați o astfel de copie este atunci când unui obiect i se alocă memorie la creare. De exemplu, să luăm două obiecte, A şi B, din aceeaşi clasă numită *TipClasa*, cărora li se alocă memorie în momentul creării. De asemenea, să presupunem că A există deja. Deci A are deja alocată memorie. Mai departe, să presupunem că A este folosit pentru a inițializa pe B, aşa cum se arată aici:

TipClasa B = A;

Dacă se face o simplă copie pe biți, atunci B va fi o copie exactă a lui A. Aceasta înseamnă că B va avea alocată exact aceeaşi zonă de memorie pe care o foloseşte A, în loc să i se aloce propria sa zonă de memorie. Este clar că nu acesta este rezultatul dorit. De exemplu, dacă *TipClasa* conține un destructor care eliberează memoria, atunci aceeaşi zonă de memorie va fi eliberată de două ori, când A şi B sunt distruse!

Aceeaşi problemă poate să apară în încă două cazuri. Primul este atunci când este făcută o copie a unui obiect pentru a fi pasată ca argument către o funcție. Al doilea este atunci când se creează un obiect temporar ca valoare returnată de către o funcție. (Amintiți-vă că obiectele temporare sunt create automat pentru a păstra valoarea întoarsă de o funcție şi că ele mai pot fi construite în anumite alte circumstanțe.)

Pentru a rezolva problema, C++ vă permite să creați un *constructor de copii*, pe care îl foloseşte compilatorul atunci când un obiect este utilizat pentru a inițializa un altul. Când există un constructor pentru copie, se renunță la copierea pe biți. Forma generală a unui constructor pentru copie este:

numeclasa(const *numeclasa* &*o*) {
    // corpul constructorului
}

Aici *o* este o referință spre obiectul din partea dreaptă a inițializării. Este permis unui constructor de copie să aibă parametri în plus, atâta timp cât ei au definite argumente implicite. Însă, în toate cazurile, primul parametru trebuie să fie o referință spre obiectul care face inițializarea.

Inițializarea apare în trei feluri: când un obiect inițializează altul, când se face o copie a unui obiect pentru a fi transmisă unei funcții sau când este generat un obiect temporar (cel mai adesea ca valoare returnată). De exemplu, fiecare dintre următoarele instrucțiuni implică inițializare:

```
clasamea x = y; // initializare
func(x); // transmitere de parametru
y = func(); // memoreaza obiectul temporar
```

În continuare se dă un exemplu în care este necesară o funcție constructor de copie explicită. Programul creează un tip de matrice „sigură" de întregi, (foarte) limitat, care verifică limitele pentru a nu fi depăşite. Memoria pentru fiecare matrice este alocată prin utilizarea operatorului **new**, iar în interiorul fiecărui obiect - matrice este păstrat un pointer spre aceasta.

```
/* Acest program creeaza o clasa de matrice "sigure".
   Deoarece spatiul pentru matrice este alocat folosind
   new, pentru a aloca memorie cand un obiect-matrice este
   folosit pentru a initializa un altul, este introdus un
   constructor de copie.
*/
#include <iostream.h>
```

```
#include <stdlib.h>

class matrice {
    int *p;
    int marime;
public:
    matrice(int mar) {
          p = new int[mar];
          if(!p) exit(1);
          marime = mar;
    }
    ~matrice() {delete [] p;}

    // constructor de copie
    matrice(const matrice &a);

    void pune(int i, int j) {
          if(i>=0 && i<marime) p[i] = j;
    }
    int da(int i) {
          return p[i];
    }
};

// constructor de copie
matrice::matrice(const matrice &a) {
    int i;

    p = new int[a.marime];
    if(!p) exit(1);
    for(i=0; i<a.marime; i++) p[i] = a.p[i];
}

main()
{
    matrice num(10);
    int i;

    for(i=0; i<10; i++) num.pune(i, i);
    for(i=9; i>=0; i--) cout << num.da(i);
    cout << "\n";

    // creeaza alta matrice si o initializeaza cu num
```



```
    matrice x=num; // apeleaza constructorul pentru copie
    for(i=0; i<10; i++) cout << x.da(i);

    return 0;
}
```

Când **num** este folosit pentru a iniţializa pe **x**, este apelat constructorul de copie, este alocată memoria pentru noua matrice şi este stocată în **x.p**, iar conţinutul din **num** este copiat în matricea lui **x**. În acest fel, **x** şi **num** conţin matrice care au aceleaşi valori, dar fiecare matrice este independentă şi distinctă. (Adică **num.p** şi **x.p** nu indică spre aceeaşi zonă de memorie.) Dacă funcţia constructor de copie nu ar fi fost creeată, iniţializarea implicită pe biţi ar fi determinat ca **x** şi **num** să utilizeze aceeaşi memorie pentru matricele lor. (Deci, **num.p** şi **x.p** ar fi indicat spre aceeaşi locaţie.)

Constructorul de copie este apelat doar pentru iniţializări. De exemplu, următoarea secvenţă nu apelează constructorul de copie definit în programul precedent.

```
matrice a(10);
    .
    .
    .
matrice b(10);
b = a; // nu apeleaza constructorul de copie
```

În acest caz, **b = a** efectuează operaţia de atribuire. Dacă = nu este supraîncărcat (şi aici nu este cazul), va fi efectuată o copie pe biţi. De aceea, în unele cazuri, pentru a evita problemele, veţi avea nevoie şi să supraîncărcaţi operatorul =, şi să creaţi un constructor pentru copie.

Iniţializare dinamică

Iniţializarea dinamică este procesul prin care variabilele sunt iniţializate în timpul rulării şi nu în timpul compilării. Mai mult, iniţializarea dinamică permite ca o variabilă să fie iniţializată folosind orice expresie validă în acel moment, inclusiv alte variabile şi apelări de funcţii. Atât C cât şi C++ permit iniţializarea dinamică a variabilelor locale. Dar, în C++, puteţi să le iniţializaţi dinamic şi pe cele globale. De exemplu, acest program este perfect valid:

```
#include <iostream.h>
#include <stdlib.h>

int i = atoi("1233"); // valida in C++, nu in C
```

```
int x = i * 2; // valida in C++, nu in C

main()
{
    cout << "valoarea lui i este " << i << endl;
    cout << "valoarea lui x este " << x << endl;
    return 0;
}
```

După cum era de aşteptat, programul afişează această ieşire:

```
valoarea lui i este 1233
valoarea lui x este 2466
```

Funcţii membre const şi volatile

Funcţiile membre ale claselor pot fi declarate ca **volatile**, **const** sau de ambele tipuri. Se aplică câteva reguli. Prima, obiectele declarate ca **volatile** pot apela doar funcţiile membre declarate, de asemenea, ca **volatile**. Un obiect de tip **const** nu poate apela o funcţie membru care nu este **const**. Dar, o funcţie membru **const** poate fi apelată atât de obiectele **const** cât şi de celelalte. O funcţie membru **const** nu poate modifica obiectul care o apelează. Pentru funcţiile care sunt atât **const** cât şi **volatile** regulile se combină.

Pentru a specifica o funcţie membru ca fiind **const** sau **volatile**, folosiţi formele prezentate în acest exemplu:

```
class X {
public:
    int f1() const; // functie membru const
    void f2(int a) volatile; // functie membru volatile
    char *f3() const volatile; // functie membru const volatile
};
```

Utilizarea cuvântului cheie asm

Puteţi să integraţi limbajul de asamblare direct în programul dvs. în C++ folosind cuvântul cheie **asm**. El are această sintaxă:

asm("*şir*");

Aici, *şir* este transmis, nemodificat, către asamblor.

Mai multe compilatoare admit trei forme generale, uşor diferite, ale instrucţiunii **asm**, ele fiind prezentate aici:

asm *instrucţiune*;
asm *instrucţiune* linie nouă
asm {
    secvenţă de instrucţiuni
}

Aici, *instrucţiune* este orice instrucţiune validă a limbajului de asamblare.

Ca un exemplu simplu (şi destul de „sigur"), acest program foloseşte **asm** pentru a executa o instrucţiune **INT 5**, care apelează funcţia print-screen:

```
//Afiseaza pe ecran.
#include <iostream.h>

main(void)
{
    asm int 5; // foloseste asm int 5
    return 0;
}
```

ATENŢIE: *Trebuie să posedaţi cunoştinţe temeinice în domeniul programării în limbajul de asamblare pentru a folosi instrucţiunea* ***asm****. Dacă nu aveţi experienţă în lucrul cu acest limbaj, este bine să evitaţi folosirea sa, deoarece pot rezulta erori foarte primejdioase.*

Specificaţii pentru editarea legăturilor

În C++ puteţi specifica modul de editare a legăturilor. De exemplu, puteţi să îi spuneţi compilatorului să editeze o funcţie ca fiind din C, din C++, sau, în funcţie de implementarea compilatorului de C++, ca fiind produsă de alt limbaj, cum ar fi FORTRAN. Implicit, funcţiilor li se editează legăturile ca fiind din C++. Dar, folosind *specificaţiile de editare a legăturilor*, puteţi să determinaţi ca unei funcţii să i se editeze legăturile ca aparţinând unui limbaj diferit. Forma generală a specificatorului de editare este:

extern "*limbaj*" *prototip-funcţie*

unde *limbaj* indică limbajul dorit. Toate compilatoarele de C++ vor admite editarea legăturilor pentru C şi pentru C++. Unele vor admite şi alte limbaje.

Următorul program determină ca **funcmeaC()** să aibă legăturile editate ca o funcție de C.

```
#include <iostream.h>

extern "C" void funcmeaC(void);

main(void)
{
    funcmeaC();
    return 0;
}

// Aceasta va avea legaturile editate ca o functie din C.
void funcmeaC(void)
{
    cout << "Acesteia i s-au editat legaturile ca pentru o
                functie din C.\n";
}
```



***NOTĂ:** Cuvântul cheie **extern** este o parte necesară pentru specificarea modului de editare de legături. Mai mult, specificarea trebuie să fie globală; ea nu poate fi folosită în interiorul unei funcții.*

Puteți specifica mai mult de o funcție o dată, folosind această formă de specificare:

extern "*limbaj*" {
prototipuri
}

Utilizarea unei specificări de editare de legături este rară și, probabil, nu veți avea nevoie să o folosiți.

Caracteristici noi adăugate de standardul ANSI C++ propus

În procesul de standardizare, comitetul ANSI C++ a adăugat mai multe caracteristici noi care nu au făcut parte din specificația originală de C++. Nu toate aceste adăugiri sunt implementate curent de orice compilator de C++ uzual. (Totuși, aceste noi facilități vor fi valabile, în viitorul apropiat, în practic toate compilatoarele de C++.) Chiar dacă nici una dintre aceste noi facilități nu sunt, practic, indispensabile pentru folosirea pe deplin a limbajului C++, unele dintre ele vă oferă un control mai bun asupra anumitor situații. Altele sunt incluse pentru comoditate. În acest paragraf, vi se oferă o scurtă trecere în revistă a acestor caracteristici. Totuși, deoarece natura lor exactă este în curs de definire, pentru detalii privind implementarea lor va trebui să verificați manualul de utilizare a compilatorului dvs.



***NOTĂ:** Deoarece standardul de C++ este încă în stadiu de dezvoltare, nu există nici o garanție că vreuna dintre caracteristicile descrise în acest paragraf va fi definită de versiunea finală a acestuia. Dar este foarte probabil că toate aceste facilități vor face parte din standardul de C++.*

Noi operatori de modelare

Chiar dacă C++ admite în întregime operatorii clasici de modelare, standardul ANSI C++ propus definește încă patru. Ei sunt **const_cast**, **dynamic_cast**, **reinterpret_cast** și **static_cast**. Formele lor generale sunt:

const_cast<*tip*>(*obiect*)
dynamic_cast<*tip*>(*obiect*)
reinterpret_cast<*tip*>(*obiect*)
static_cast<*tip*>(*obiect*)

Aici, *tip* specifică tipul final al modelatorului, iar *obiect* este obiectul care este modelat în noul tip.

Operatorul **const_cast** este folosit pentru a înlătura atributele **const** și/sau **volatile**. Tipul final trebuie să fie același cu tipul sursă, cu excepția modificării caracteristicilor **const** și **volatile**. Cea mai uzuală întrebuințare pentru **const_cast** este eliminarea specificației **const**.

dynamic_cast efectuează o modelare în timpul rulării care verifică validitatea unei modelări. Dacă modelarea nu poate fi făcută, ea eșuează, iar expresia este evaluată ca null. Utilizarea sa principală este pentru modelări asupra tipurilor polimorfice. (Clasele polimorfice sunt clase care conțin funcții virtuale.) De exemplu, **dynamic_cast** poate să returneze un pointer către un obiect derivat, dându-se un pointer către o clasă de bază polimorfică. Dacă obiectul indicat nu este un obiect al clasei de bază sau al unei clase derivate, atunci **dynamic_cast** se evaluează ca null.

Operatorul **static_cast** efectuează o modelare nepolimorfică. De exemplu, el poate fi folosit pentru a modela un pointer al clasei de bază într-unul al unei clase derivate. El poate fi utilizat, de asemenea, pentru orice conversie standard. Operatorul **reinterpret_cast** modifică un tip într-unul fundamental diferit. De

exemplu, el poate fi folosit pentru a schimba un pointer într-un întreg. Un **reinterpret_cast** trebuie utilizat pentru modelarea tipurilor inerent incompatibile.

Doar **const_cast** poate să înlăture atributul **const**. Aceasta înseamnă că nici **dynamic_cast**, nici **static_cast** şi nici **reinterpret_cast** nu pot modifica o caracteristică de tip **const**.

Următorul program ilustrează utilizarea operatorului **reinterpret_cast**.

```
// Un exemplu care foloseste reinterpret_cast.
#include <iostream.h>

main()
{
    int i;
    char *p = "Acesta este un sir";

    i = reinterpret_cast<int> (p); // transforma un pointer
                                   // in intreg

    cout << i;

    return 0;
}
```

Tipul de date bool

Deşi, practic, nu este necesar, standardul ANSI C++ propus a adăugat tipul de date **bool**, care este capabil să păstreze o valoare booleană. Obiectele de tip **bool** pot avea doar valorile **true** (adevărat) sau **false** (fals). (**true** şi **false** sunt cuvinte-cheie care fac acum parte din limbajul C++.) Valorile de tip **bool** sunt ridicate automat la întregi, atunci când sunt folosite cu expresii ne-booleene.

Utilizarea unui nume pentru zona de influenţă

Standardul ANSI C++ propus a adăugat şi cuvântul cheie **namespace**, care poate fi folosit pentru a defini o sferă de acţiune. Iată, aici, forma generală pentru **namespace**:

namespace *nume* {
    // declarări de obiecte
}

De exemplu,


```
namespace NumeleZonei {
    int i, k;
    void funcmea(int j) { cout << j; }
}
```

Aici **i**, **k** şi **funcmea()** fac parte din sfera de acţiune definită de **NumeleZonei**.

Deoarece un nume al domeniului de influenţă defineşte o sferă de acţiune, pentru a vă referi la obiectele definite în interiorul său, va trebui să folosiţi operatorul de specificare a domeniului. De exemplu, pentru a atribui lui **i** valoarea 10, trebuie să folosiţi instrucţiunea:

```
NumeleSpatiu::i = 10;
```

Dacă membrii zonei de influenţă vor fi folosiţi frecvent, puteţi folosi o directivă **using** pentru a simplifica accesul la ei. Instrucţiunea **using** are două forme generale:

using namespace *nume*;
using *nume::membru*;

În prima formă, *nume* specifică numele pentru zona de influenţă la care doriţi acces. Toţi membrii care sunt definiţi în interiorul zonei indicate pot fi folosiţi fără specificator. În cea de-a doua formă, devine vizibil doar un obiect, acela menţionat explicit. De exemplu, presupunând **NumeZona** prezentat anterior, sunt valide următoarele instrucţiuni **using** şi atribuiri:

```
using NumeZona::k; // este vizibil doar k
k = 10; // corect, deoarece k este vizibil
using namespace NumeZona; // sunt vizibili toti membrii din
                          // NumeleZonei
i = 10; // OK deoarece acum sunt vizibili toti membrii din
        // NumeleZonei
```

*NOTĂ: Forma exactă şi natura operatorului **namespace** şi ale instrucţiunilor **using** sunt în curs de finalizare. Pentru detalii de implementare relativ la aceste două caracteristici, verificaţi manualul compilatorului dvs.*

Identificarea tipului în timpul rulării

Una dintre cele mai importante facilităţi adăugate de standardul ANSI C++ propus este *identificarea tipului în timpul rulării* (RTTI). Utilizând acest tip de identificare, puteţi să determinaţi tipul unui obiect în timpul executării programului. Pentru a se

obţine tipul unui obiect, se foloseşte **typeid**. Ca să îl puteţi folosi, trebuie să includeţi fişierul antet TYPEINFO.H. Forma sa generală este:

typeid(*obiect*)

obiect este aici obiectul al cărui tip doriţi să-l aflaţi. **typeid** returnează o referinţă către un obiect de tipul **type_info** care descrie tipul obiectului definit prin *obiect*. Clasa **type_info** defineşte următorii membri publici:

```
bool operator==(const type_info &ob) const;
bool operator!=(const type_info &ob) const;
bool before(const type_info &ob) const;
const char *name() const;
```

Compararea tipurilor este asigurată de == şi de != supraîncărcate. Funcţia **before()** returnează adevărat dacă obiectul care apelează se află înaintea obiectului folosit ca parametru, în ordinea citirii. (Această funcţie este, de obicei, doar pentru uz intern. Valoarea sa returnată nu are nici o legătură cu moştenirea sau cu ierarhizarea claselor.) Funcţia **name()** returnează un pointer către numele tipului. (Deoarece **typeid** este în curs de a fi definit, pentru detalii privind orice alte tipuri de operaţii definite de clasa **type_info**, va trebui să consultaţi manualul compilatorului dvs.)

Când **typeid** este aplicat unui pointer către clasa de bază al unei clase polimorfice, el va returna automat tipul obiectului către care indică, inclusiv numele oricărei clase derivate din acea bază. (Amintiţi-vă că o clasă polimorfică este una ce conţine cel puţin o funcţie virtuală.)

typeid este ilustrat de către următorul program:

```
// Un exemplu care foloseste typeid.
#include <iostream.h>
#include <typeinfo.h>

class ClasaBaza {
    int a, b;
    virtual void f() {}; // face ClasaBaza polimorfica
};

class Derivat1: public ClasaBaza {
    int i, j;
};

class Derivat2: public ClasaBaza {
    int k;
};

main()
{
    int i;
    ClasaBaza *p, obbaza;
    Derivat1 ob1;
    Derivat2 ob2;

    // Mai intai afiseaza numele tipului pentru un tip
    // incorporat.
    cout << "Tipul lui i este ";
    cout << typeid(i).name() << endl;

    // Prezinta typeid cu tipuri polimorfice.
    p = &obbaza;
    cout << "p indica spre un obiect de tipul ";
    cout << typeid(*p).name() << endl;

    p = &ob1;
    cout << "p indica spre un obiect de tipul ";
    cout << typeid(*p).name() << endl;

    p = &ob2;
    cout << "p indica spre un obiect de tipul ";
    cout << typeid(*p).name() << endl;

    return 0;
}
```

Iată ieşirea produsă de acest program:

```
Tipul lui i este int
p indica spre un obiect de tipul ClasaBaza
p indica spre un obiect de tipul Derivat1
p indica spre un obiect de tipul Derivat2
```

După cum s-a menţionat, când **typeid** este aplicat unui pointer al clasei de bază de tip polimorfic, tipul obiectului spre care indică va fi determinat în timpul rulării, după cum arată ieşirea produsă de către program.

Identificarea tipului în timpul rulării nu este ceva ce se foloseşte în orice program. Totuşi, când lucraţi cu tipuri polimorfice, ele vă permit să cunoaşteţi asupra cărui tip de obiect se operează într-o situaţie dată.

Constructori expliciţi

Standardul propus pentru ANSI C++ a definit cuvântul cheie **explicit**. El este folosit pentru a crea „constructori fără conversie". De exemplu, dându-se următoarea clasă,

```
class ClasaMea {
    int i;
public:
    ClasaMea(int j) {i = j;}
    // ...
};
```

obiectele din **ClasaMea** pot fi declarate astfel:

```
ClasaMea ob1(1);
ClasaMea ob2 = 10;
```

În acest caz, instrucţiunea

```
ClasaMea ob2 = 10;
```

este convertită automat în forma

```
ClasaMea ob2(10);
```

Dar, dacă funcţia constructor pentru **ClasaMea** este declarată **explicit**, această conversie automată nu va fi efectuată. Iată **ClasaMea** folosind un constructor **explicit**:

```
class ClasaMea {
    int i;
public:
    explicit ClasaMea(int j) {i = j;}
    // ...
};
```

Acum vor fi permişi doar constructori de forma:

```
ClasaMea ob(110);
```


Utilizarea operatorului mutable

Standardul ANSI C++ propus defineşte cuvântul cheie **mutable**. El este folosit pentru a permite ca un membru al unui obiect să suprascrie atributul **const**. Cu alte cuvinte, un membru **mutable** al unui obiect **const** nu este constant şi poate fi modificat.

Tipul wchar_t

Standardul ANSI propus pentru C++ defineşte tipul **wchar_t** care poate conţine caractere foarte lungi. De obicei ele sunt memorate pe 16 biţi şi sunt folosite pentru a reprezenta seturile de caractere ale limbajelor care au mai mult de 255 de caractere.

Fişiere antet noi

Propunerea pentru standardul ANSI C++ a definit o nouă cale de a introduce fişierele antet. Totuşi, stilul clasic (care este folosit de această carte) este admis pe deplin. Noul stil nu cere utilizarea efectivă a unui nume de fişier. În schimb, este folosit un specificator standard pentru antet, care, dacă este necesar, va fi transformat de către compilator într-un nume de fişier. De exemplu, noul stil al formatului de antet pentru includerea fişierelor sistemului de I/O este:

```
#include <iostream>
```

După cum puteţi vedea, grupul **.h** a fost lansat. Regula poate fi generalizată. De exemplu, folosind noul stil pentru formatul fişierului antet, următoarea instrucţiune include antetul pentru sistemul de I/O cu fişiere:

```
#include <fstream>
```

Va trebui să verificaţi în manualul compilatorului dacă acesta admite noul stil pentru specificarea fişierelor antet.

Diferenţe între C şi C++

În cea mai mare parte, C++ este un superstandard ANSI C şi, teoretic, toate programele în C sunt şi programe în C++. Totuşi, există câteva diferenţe, cele mai importante dintre ele fiind discutate aici.

În C++ variabilele locale pot fi declarate oriunde în interiorul unui bloc. În C, ele trebuie declarate la începutul blocului, înainte să apară orice instrucţiune de „acţiune".

Una dintre cele mai importante, deşi subtile, diferenţe dintre C şi C++ este că în C o funcţie declarată astfel

```
int f();
```

nu spune *nimic* despre nici un parametru al acesteia. Deci, atunci când între parantezele care urmează numelui funcţiei nu este specificat nimic, în C înseamnă că nu s-a afirmat nimic despre nici un parametru al acelei funcţii. Ea poate să aibă parametri, poate să nu aibă. Dar, în C++ o astfel de declaraţie de funcţie înseamnă că ea *nu* are parametri. Deci, în C++, aceste două declaraţii sunt echivalente:

```
int f();
int f(void);
```

În C++, **void** este opţional. Mulţi programatori de C++ includ **void** pentru a fi foarte clar pentru oricine citeşte un program în care o funcţie nu are nici un parametru, dar practic el nu este necesar.

În C++ toate funcţiile trebuie să aibă prototip. Acest lucru este opţional în C (chiar dacă o bună practică de programare recomandă folosirea deplină a prototipurilor într-un program în C).

O mică dar potenţial importantă diferenţă între C şi C++ este că în C, o constantă caracter este ridicată automat la rangul unui întreg. În C++ nu este aşa.

În C nu este o eroare să se declare o variabilă globală de mai multe ori, chiar dacă nu este o practică de programare prea bună. În C++ aceasta este o eroare.

În C, un identificator poate să aibă lungimea de maximum 31 de caractere. În C++ nu există astfel de limite. Totuşi, din punct de vedere practic, identificatorii extrem de lungi sunt incomozi şi rareori necesari.

În C, deşi nu se obişnuieşte, puteţi să apelaţi **main()** din interiorul programului dvs. Acest lucru nu este permis în C++.

În C nu puteţi să preluaţi adresa unei variabile de tip **register**. În C++ acest lucru este permis.

Partea a III-a

Câteva aplicaţii de C++

Partea a treia a acestei cărţi conţine câteva exemple de aplicaţii scrise în C++. Scopul acestei secţiuni este dublu. În primul rând, exemplele ajută la ilustrarea beneficiilor programării orientate pe obiecte, inclusiv avantajele polimorfismului, ale încapsulării şi moştenirii şi ale creării bibliotecilor de clase. Apoi, exemplele arată cum poate fi aplicat C++ pentru a rezolva diverse tipuri de probleme - orientate sau nu pe obiecte. Amintiţi-vă că C++ este o versiune îmbunătăţită şi lărgită de C, care oferă programatorului mai multă putere şi flexibilitate, independent de metodologia orientării pe obiecte. Deci, nu are neapărat importanţă dacă veţi folosi C++ pentru a realiza OOP sau doar pentru a vă ridica ştacheta peste nivelul normal al temelor dvs. de programare.

Capitolul 23

O clasă de tip şir

C++

După cum ştiţi, şirurile sunt introduse în C++ ca matrice de caractere terminate cu null şi nu ca tipuri de date separate. Această abordare face ca şirurile din C++ să fie puternice, elegante şi eficiente. De asemenea, relaţia strânsă dintre matrice şi pointeri vă permite să scrieţi multe operaţii cu şiruri „scurte şi cu miez". Totuşi, de multe ori este necesar să folosiţi un şir, dar nu vă trebuie să stabiliţi un grad înalt de eficienţă şi de putere. În aceste cazuri, lucrul cu şiruri în C++ poate să devină o adevărată corvoadă. Din fericire, este posibil ca în C++ să creaţi un tip de şir care pierde oarecum din eficienţă, dar câştigă mult în uşurinţa utilizării.

În acest capitol este dezvoltată o clasă de tip şir, care face mult mai uşoară crearea, utilizarea şi manevrarea şirurilor.



NOTĂ: În momentul scrierii acestei cărţi, comitetul de standardizare ANSI C++ este în procesul de definire a unei clase standard de tip şir. (Dar, forma sa finală nu a fost încă stabilită.) Scopul acestui capitol nu este să prezinte o alternativă a acestei clase, ci de a vă oferi o imagine a uşurinţei în care orice tip nou de date poate fi adăugat şi integrat în mediul C++. Crearea unei clase de tip şir este un exemplu esenţial al acestui proces. Pe lângă faptul că noua clasă de tip şir din acest capitol este mult mai simplă decât cea din C++ standard, mai există un avantaj: ea vă oferă controlul complet al modului în care sunt implementate şi manevrate şirurile. O veţi găsi probabil folositoare în multe situaţii.

Definirea unui tip de şir

În primul rând este important să definim ce se înţelege printr-un tip de şir şi ce fel de operaţii pot fi efectuate cu el. Din fericire, alte limbaje au definit tipurile de şir şi pot fi luate ca modele pe care ne bazăm descrierea acestora. La prima vedere poate să pară ciudat, dar un model foarte bun pentru introducerea tipului de şiruri este BASIC. Chiar dacă majoritatea programatorilor de C++ nu sunt entuziasmaţi de acesta ca limbaj de programare în general, modul în care manevrează şiruri este intuitiv şi uşor de folosit. BASIC este, de asemenea, un model bun, deoarece îl cunosc practic toţi programatorii.

Clasa de tip şir prezentată în acest capitol nu copiază exact abordarea din BASIC, dar împrumută cele mai importante facilităţi, care vor fi studiate în continuare.

În BASIC, pentru a da o valoare unui şir, îi atribuiţi pur şi simplu un şir între ghilimele folosind operatorul de atribuire, =. De exemplu, aceasta este o atribuire validă în BASIC:

```
A$ = "Acesta este un sir"
```

(Toate variabilele de tip şir din BASIC trebuie să se termine cu un semn dolar. Desigur, clasa de tip şir prezentată în acest capitol nu va avea această restricţie.) Instrucţiunea de mai sus atribuie variabilei **A$** şirul **"acesta este un sir"**.

De asemenea, puteţi să atribuiţi o variabilă şir altei variabile şir. De exemplu, următoarea comandă copiază în **B$** şirul conţinut în **A$**.

```
B$ = A$
```

După cum puteţi vedea, diferenţa principală dintre BASIC şi C++ în ce priveşte atribuirea unui şir pentru o variabilă de tip şir este că BASIC foloseşte un operator, în timp ce C++ foloseşte o apelare a funcţiei **strcpy()** (deşi C++ permite ca matricele de tip caracter să fie *iniţializate* cu operatorul =).

Operatorul + este folosit în BASIC pentru a concatena două şiruri. De exemplu, această secvenţă face ca **C$** să conţină valoarea „Va salut".

```
A$ = "Va "
B$ = "salut"
C$ = A$ + B$
```

De fapt, secvenţa precedentă ar putea fi simplificată astfel:

```
A$ = "Va "
C$ = A$ + "salut"
```

Aici, o variabilă de tip şir este concatenată cu un şir cuprins între ghilimele. Astfel, BASIC permite unei variabile de tip şir să fie concatenată cu altă variabilă sau cu şiruri cuprinse între paranteze.

O diferenţă între concatenarea unui şir în BASIC şi funcţia standard **strcat()** este că aceasta din urmă modifică unul dintre şirurile apelate astfel încât să conţină rezultatul. În schimb, când alipiţi două şiruri în BASIC, apare un şir temporar care conţine concatenarea, iar şirurile originale rămân nemodificate.

Comparaţiile de şiruri în BASIC sunt intuitive deoarece folosesc aceiaşi operatori relaţionali folosiţi şi când se compară alte tipuri de date. Toate comparaţiile de şiruri sunt efectuate în ordine alfabetică. De exemplu, aceasta determină dacă **A$** este mai mare ca **B$**.

```
IF A$ > B$ THEN PRINT "A$ este mai mare ca B$"
```

După cum arată exemplul precedent, avantajul principal al abordărilor şirurilor C din BASIC este că el permite ca toate operaţiile importante să fie efectuate cu aceiaşi operatori folosiţi pentru alte tipuri de date. Într-un fel, BASIC supraîncarcă atribuirea, adunarea şi operatorii relaţionali astfel încât să poată lucra şi cu şiruri. Acesta este conceptul de bază care a fost introdus în acest capitol de către clasele

de tip şir. Tipul de şir introdus aici va înlocui apelările funcţiilor de bibliotecă cu operatori supraîncărcaţi.

Cunoscând abordarea şirurilor din BASIC, suntem acum gata să creăm o clasă de tip şir în C++.

Clasa StrType

Clasa de tip şir definită aici va îndeplini următoarele cerinţe:

- Şirurilor le pot fi atribuite valori cu operatorul de atribuire.
- Obiectelor de tip şir le pot fi atribuite alte obiecte de tip şir sau şiruri între ghilimele.
- Concatenarea a două obiecte de tip şir se realizează cu operatorul +.
- Operatorul - poate fi folosit pentru a exclude un subşir dintr-un şir.
- Comparările de şiruri sunt efectuate cu operatorii relaţionali.
- Un obiect de tip şir poate fi iniţializat ori folosind şiruri între ghilimele ori alte obiecte de tip şir.
- Şirurile trebuie să fie de lungimi arbitrare şi variabile. Aceasta implică pentru fiecare şir alocarea dinamică a memoriei.
- Va fi asigurată o metodă de convertire a obiectelor de tip şir în şiruri cu terminaţia null.

Clasa care va trata şirurile se numeşte **StrType**. Iată declararea ei:

```
class StrType {
    char *p;
    int marime;
public:
    StrType(char *sir);
    StrType();
    StrType(const StrType &o); // constructor de copie
    ~StrType() {delete [] p;}

    friend ostream &operator<<(ostream &stream, StrType &o);
    friend istream &operator>>(istream &stream, StrType &o);

    StrType operator=(StrType &O); // atribuie un obiect
                                   // tip StrType
    StrType operator=(char *s); // atribuie un sir intre
                                // ghilimele
```



```
    StrType operator+(StrType &o); /* concateneaza un obiect
                                      tip StrType */
    StrType operator+(char *s); /* concateneaza un sir intre
                                   ghilimele */
    friend StrType operator+(char *s, StrType &o); /* con-
              cateneaza un sir intre ghilimele cu un obiect
              tip StrType */

    StrType operator-(StrType &O); // exclud un subsir
    StrType operator-(char *s); /* exclud un subsir intre
                                   ghilimele */

    // operatii relationale intre obiecte de tip StrType
    int operator==(StrType &o); {return !strcmp(p, o.p);}
    int operator!=(StrType &o); {return strcmp(p, o.p);}
    int operator<(StrType &o); {return strcmp(p, o.p) < 0;}
    int operator>(StrType &o); {return strcmp(p, o.p) > 0;}
    int operator<=(StrType &o); {return strcmp(p, o.p) <= 0;}
    int operator>=(StrType &o); {return strcmp(p, o.p) >= 0;}

    // operatii intre obiecte de tip StrType si siruri
    // intre ghilimele
    int operator==(char *s) {return !strcmp(p, s);}
    int operator!=(char *s) {return strcmp(p, s);}
    int operator<(char *s) {return strcmp(p, s) < 0;}
    int operator>(char *s) {return strcmp(p, s) > 0;}
    int operator<=(char *s) {return strcmp(p, s) <= 0;}
    int operator>=(char *s) {return strcmp(p, s) >= 0;}

    int strsize() {return strlen(p);} /* returneaza
                                         lungimea sirului */
    void makestr(char *s) {strcpy(s, p);} // creeaza siruri
                                          // intre ghilimele
    operator char *() { return p;} // conversie in char *
};
```

Secţiunea particulară din **StrType** conţine doar două elemente: **p** şi **marime**. Când este creat un obiect de tip şir, memoria pentru păstrarea şirului este alocată dinamic de către **new**, iar în **p** este pus un pointer spre acea memorie. Şirul spre care indică **p** va fi unul normal, o matrice de tip caracter terminată cu un caracter de null. Chiar dacă nu este, practic, necesar, mărimea şirului este păstrată în **marime**. Deoarece şirul spre care indică **p** este unul normal, ar fi posibil să se

calculeze mărimea sa de câte ori este necesar. Totuşi, după cum veţi vedea, această valoare este folosită atât de des de către funcţiile membre din **StrType**, încât apelările repetate ale funcţiei **strlen()** nu sunt justificate.

Următoarele paragrafe detaliază cum lucrează clasa **StrType**.

Funcţiile constructor şi destructor

Un obiect de tip **StrType** poate fi declarat în trei feluri. El poate fi declarat fără nici o iniţializare ori iniţializat cu un şir între ghilimele sau cu un obiect de tip **StrType**. Constructorii care admit aceste trei operaţii sunt prezentaţi aici:

```
// Nici o initializare explicita.
StrType::StrType() {
   marime = 1; // face loc pentru terminatia de null
   p = new char[marime];
   if(!p) {
         cout << "Eroare de alocare\n";
         exit(1);
   }
   *p = '\0';
}

// Initializare folosind un sir intre ghilimele, terminat
// cu null.
StrType::StrType(char *sir) {
   marime = strlen(sir) + 1; // face loc pentru
                             // terminatorul null
   p = new char[size];
   if(!p) {
         cout << "Eroare de alocare\n";
         exit(1);
   }
   strcpy(p, sir);
}

// Initializare folosind un obiect StrType
StrType::StrType(const StrType&o) {
   marime = o.marime;
   p = new char[marime];
   if(!p) {
         cout << "Eroare de alocare\n";
         exit(1);
   }
   strcpy(p, o.p);
}
```


Când este creat un obiect tip **StrType** fără iniţializare, i se atribuie valoarea unui şir null. Deşi şirul s-ar putea să fi fost fi lăsat nedefinit, faptul că toate obiectele de tip **StrType** conţin un şir valid, terminat cu null, simplifică multe alte funcţii membre.

Când un obiect din clasa **StrType** este iniţializat printr-un şir între ghilimele, mai întâi se determină lungimea şirului. Valoarea sa este memorată în **marime**. Apoi, se alocă memoria necesară prin **new**. După ce verificarea de siguranţă confirmă că **p** nu este nul, şirul de iniţializare este copiat în zona de memorie spre care indică **p**.

Când este folosit un obiect de tip **StrType** pentru a iniţializa un altul, procesul este similar cu cel al utilizării unui şir între ghilimele. Singura diferenţă este aceea că mărimea unui şir este cunoscută şi nu mai trebuie calculată. O astfel de versiune a constructorului clasei **StrType** este şi constructorul de copii al clasei. Acesta va fi apelat ori de câte ori este folosit un obiect de tip **StrType** pentru a iniţializa un altul. Cu alte cuvinte, va fi apelat când sunt create obiecte temporare şi când obiectele de tip **StrType** sunt pasate spre funcţii. (Pentru constructorii de copii vedeţi **Capitolul 22**.)

Dându-se constructorii anteriori, sunt permise următoarele declaraţii:

```
StrType x("sirul meu"); // foloseste sir intre ghilimele
StrType y(x); // foloseste un alt obiect
StrType z; // nici o initializare explicita
```

Funcţia destructor pentru **StrType** eliberează pur şi simplu memoria spre care indică **p**.

I/O cu şiruri

Deoarece este foarte uzual să doriţi să introduceţi sau să obţineţi şiruri, clasa **StrType** supraîncarcă operatorii << şi >>, aşa cum se prezintă aici;

```
// Scrie un sir la iesire
ostream &operator<<(ostream &stream, StrType &o)
{
    stream << o.p;
    return stream;
}

// Introduce un sir.
istream &operator>>(istream &stream, StrType &o)
```

```
{
    char t[255]; // marime arbitrara - modificati-o daca
                 // este necesar
    int lung;

    for(lung=0; lung<255; lung++) {
          stream.da(t[lung]);
          if(t[lung])=='\n') break;
          if(t[lung]=='\b')
          if(lung) {
                lung--;
                cout << "'\b'";
          }
    }
    t[lung] = '\0';
    lung++;

    if(lung > o.marime) {
          delete o.p;
          o.p = new char[lung];
          if(!o.p) {
                cout << "Eroare de alocare\n";
                exit(1);
          }
          o.marime = lung;
    }
    strcpy(o.p, t);
    return stream;
}
```

După cum puteți vedea, metoda este foarte simplă. Totuși, rețineți că parametrul **o** este transmis prin referință. Deoarece obiectele de tip **StrType** pot fi foarte mari, transmiterea lor prin referință este mai eficientă decât cea prin valoare. Din acest motiv, toți parametrii din **StrType** sunt transmiși prin referință. (Orice funcție pe care o creați și care preia parametri din **StrType** va face, probabil, la fel.)

Introducerea șirurilor se dovedește a fi puțin mai dificilă decât obținerea lor, deoarece nu se poate utiliza o instrucțiune ca următoarea pentru a se citi intrarea.

```
stream >> t;
```

Motivul este că operația de intrare normală care citește șirurile în stilul C++ se oprește din citit când întâlnește primul caracter de spațiu liber. De aceea, este

necesar să citiți un șir introducând caracterele unul câte unul. Versiunea operatorului >> supraîncărcat pentru tipul **StrType** citește caractere până când se întâlnește o linie nouă.

O dată ce șirul a fost citit, dacă mărimea noului șir depășește pe cea păstrată curent în **o**, acea memorie este eliberată și este alocată o cantitate mai mare. Apoi noul șir este copiat aici.

Funcțiile de atribuire

Puteți să atribuiți valori unui obiect de tip **StrType** în două feluri. În primul rând, puteți să îi atribuiți un alt obiect de tip **StrType**. În al doilea rând, puteți să îi atribuiți un șir între ghilimele. Iată aici cele două funcții **operator=()** supraîncărcate care realizează aceste operații:

```
// Atribuie un obiect de tip StrType unui obiect de tip
// StrType.
StrType StrType::operator=(StrType &o)
{
    StrType temp(o.p);

    if(o.marime > marime) {
          delete p; // elibereaza memoria veche
          p = new char[o.marime];
          marime = o.marime;
          if(!p) {
                cout << "Eroare de alocare\n";
                exit(1);
          }
    }

    strcpy(p, o.p);
    strcpy(temp.p, o.p);

    return temp;
}

// Atribuie un sir intre ghilimele unui obiect de tip
// StrType.
StrType StrType::operator=(char *s)
{
    int lung = strlen(s) + 1;
    if(marime < lung) {
          delete p;
```

```
            p = new char[lung];
            marime = lung;
            if(!p) {
                  cout << "Eroare de atribuire\n";
                  exit(1);
            }
      }
      strcpy(p, s);
      return *this;
}
```

Aceste două funcţii efectuează o primă verificare pentru a vedea dacă memoria spre care indică **p**, având ca ţintă obiectul tip **StrType**, este suficient de mare pentru a păstra ceea ce doriţi să copiaţi în ea. Dacă nu este aşa, memoria veche este eliberată şi se alocă una nouă. Apoi şirul este copiat în obiect şi se returnează rezultatul. Aceste funcţii permit următoarele tipuri de atribuire:

```
StrType x("test"), y;
y = x; // obiect StrType in obiect StrType
x = "un nou sir pentru x"; // sir intre ghilimele in obiect
                           // StrType
```

Fiecare funcţie de atribuire trebuie să returneze valoarea atribuită (valoarea din partea dreaptă), astfel încât să fie admise atribuiri multiple, ca aceasta:

```
StrType x, y, z;
x = y = z = "test";
```

Concatenarea

Concatenarea a două şiruri se realizează cu operatorul +. Clasa **StrType** permite următoarele trei situaţii de concatenare distincte:

- Concatenarea unui obiect de tip **StrType** cu un alt obiect **StrType**
- Concatenarea unui obiect de tip **StrType** cu un şir între ghilimele
- Concatenarea unui şir între ghilimele cu un obiect de tip **StrType**

Când este folosit în astfel de situaţii, operatorul + produce ca rezultate de ieşire un obiect de tip **StrType** care reprezintă concatenarea celor doi operanzi. El nu modifică nici unul dintre aceştia. (Această abordare diferă de funcţia **strcat()**, care modifică primul său argument.)

Iată prezentate funcţiile **operator+()** supraîncărcate:

```
// Concateneaza doua obiecte de tip StrType.
StrType StrType::operator+(StrType &o)
{
    int lung;
    StrType temp;

    delete temp.p;
    lung = strlen(o.p) + strlen(p) + 1;
    temp.p = new char[lung];
    temp.marime = lung;
    if(!temp.p) {
          cout << "Eroare de alocare\n";
          exit(1);
    }
    strcpy(temp.p, p);

    strcat(temp.p, o.p);

    return temp;
}

// Concateneaza un obiect de tip StrType cu un sir intre
// ghilimele.
StrType StrType::operator+(char *s)
{
    int lung;
    StrType temp;

    delete temp.p;

    lung = strlen(s) + strlen(p) + 1;
    temp.p = new char[lung];
    temp.marime = lung;
    if(!temp.p) {
          cout << "Eroare de alocare\n";
          exit(1);
    }
    strcpy(temp.p, p);

    strcat(temp.p, s);
```

```
    return temp;
}

// Concateneaza un sir intre ghilimele cu un obiect tip
// StrType.
StrType operator+(char *s, StrType &o)
{
    int lung;
    StrType temp;

    delete temp.p;

    lung = strlen(s) + strlen(o.p) + 1;
    temp.p = new char[lung];
    temp.marime = lung;
    if(!temp.p) {
          cout << "Eroare de alocare\n";
          exit(1);
    }
    strcpy(temp.p, s);

    strcat(temp.p, o.p);

    return temp;
}
```

Toate trei funcţiile lucrează, în esenţă, în acelaşi fel. Mai întâi este creat un obiect de tip **StrType** numit **temp**. Acest obiect va conţine rezultatul unei concatenări şi este obiectul returnat de către funcţie. Apoi este eliberată memoria spre care indică **temp.p**. Motivul acestei operaţii este că, atunci când este creat **temp**, se alocă doar un octet de memorie (pentru a reţine un loc), deoarece nu există o iniţializare explicită. Apoi, se alocă suficientă memorie pentru a păstra concatenarea celor două şiruri. În final, cele două şiruri sunt copiate în memoria spre care indică **temp.p** şi este returnat **temp**.

Excluderi de subşiruri

O funcţie utilă pentru şiruri, care nu se găseşte în multe limbaje, este cea de excludere a subşirurilor. Aşa cum este introdusă în clasa **StrType**, funcţia de *excludere de subşiruri* elimină toate apariţiile unui subşir specificat dintr-un alt şir. Ea este realizată cu operatorul -.

Clasa **StrType** admite două cazuri de excludere de subşiruri. Unul permite ca un obiect de tip **StrType** să fie exclus din alt obiect de acelaşi tip. Celălalt face

posibilă eliminarea unui şir între ghilimele dintr-un obiect de tip **StrType**. În următorul exemplu sunt prezentate cele două funcţii **operator-()**:

```
// Exclude un subsir dintr-un sir folosind obiecte de tip
// StrType.
StrType StrType::operator-(StrType &subsir)
{
    StrType temp(p);
    char *s1;
    int i, j;

    s1 = p;
    for(i=0; *s1; i++) {
          if(*s1!=*subsir.p) { // daca nu este prima litera
                               // a subsirului
                temp.p[i] = *s1; // atunci copiaza in temp
                s1++;
          }
          else { // poate fi extras
                for(j=o; subsir.p[j]==s1[j] && subsir.p[j];
                      j++);
                if(!subsir.p[j]) { // este subsirul, deci
                                   // trebuie eliminat
                      s1 += j;
                      i--;
                }
                else { // nu este subsirul, continua copierea
                      temp.p[i] = *s1;
                      s1++;
                }
          }
    }
    temp.p[i] = '\0';
    return temp;
}

// Exclude siruri intre ghilimele din obiecte de tip
// StrType.
StrType StrType::operator-(char *subsir)
{
    StrType temp(p);
    char *s1;
    int i, j;
```

```
    s1 = p;
    for(i=0; *s1; i++) {
          if(*s1!=*subsir.p) { // daca nu este prima litera
                             // din subsir
                temp.p[i] = *s1; // atunci copiaza in temp
                s1++;
          }
          else {
                for(j=o; subsir.p[j]==s1[j] && subsir.p[j];
                     j++);
                if(!subsir.p[j]) { // este subsirul, deci
                                 // trebuie eliminat
                      s1 += j;
                      i--;
                }
                else { // nu este subsirul, continua copierea
                      temp.p[i] = *s1;
                      s1++;
                }
          }
    }
    temp.p[i] = '\0';
    return temp;
}
```

Aceste funcţii lucrează prin copierea în **temp** a conţinutului operandului stâng, eliminând în timpul procesului orice apariţie a subşirului specificat de operandul drept. Este returnat obiectul rezultat de tip **StrType**. Se înţelege că, în timpul procesului, nu este modificat nici un operand.

Clasa **StrType** permite excluderi de subşiruri ca acestea:

```
StrType x("Imi place C++"), y("place");
StrType z;

z = x - y; // z va contine "Imi C++"
z = x - "C++"; // z va contine "Imi place"

// sunt eliminate aparitii multiple
z = "ABCDABCD";
x = z - "A"; // contine "BCDBCD"
```


Operatorii relaţionali

Clasa **StrType** admite ca şirurilor să le fie aplicată întreaga gamă a operatorilor relaţionali. Operatorii relaţionali sunt redefiniţi (overload) în interiorul declarării clasei **StrType**. Pentru claritate ei sunt repetaţi aici:

```
// operatii relationale intre obiecte de tip StrType
int operator==(StrType &o)  {return !strcmp(p, o.p);}
int operator!=(StrType &o)  {return strcmp(p, o.p);}
int operator<(StrType &o)  {return strcmp(p, o.p) < 0;}
int operator>(StrType &o)  {return strcmp(p, o.p) > 0;}
int operator<=(StrType &o)  {return strcmp(p, o.p) <= 0;}
int operator>=(StrType &o)  {return strcmp(p, o.p) >= 0;}

// operatii intre obiecte de tip StrType si siruri intre
// ghilimele
int operator==(char *s) {return !strcmp(p, s);}
int operator!=(char *s) {return strcmp(p, s);}
int operator<(char *s) {return strcmp(p, s) < 0;}
int operator>(char *s) {return strcmp(p, s) > 0;}
int operator<=(char *s) {return strcmp(p, s) <= 0;}
int operator>=(char *s) {return strcmp(p, s) >= 0;}
```

Operaţiile relaţionale sunt foarte intuitive; nu ar trebui să aveţi probleme cu înţelegerea definiţiilor lor. Totuşi, reţineţi că **StrType** introduce doar comparaţia între două obiecte de acest tip sau între un obiect de tip **StrType** ca operand stâng şi un şir între ghilimele ca operand drept. Dacă vreţi să puteţi scrie şirul aflat între ghilimele în partea stângă, iar obiectul de tip **StrType** în partea dreaptă, va trebui să mai adăugaţi funcţii relaţionale.

Dându-se funcţiile supraîncărcate pentru operatori relaţionali redefinite de **StrType**, sunt permise următoarele tipuri de comparare a şirurilor:

```
StrType x("unu"), y("doi"), z("trei");

if(x < y) cout << "x mai mic decit y";

if(z=="trei") cout << "z egal trei";
y = "u";
z = "nu";
if(x==(y+z)) cout << "x este egal cu y+z";
```

Diverse funcţii pentru şiruri

Clasa **StrType** defineşte trei funcţii care fac ca obiectele de tip **StrType** să se integreze şi mai bine în rândul şirurilor obişnuite din C++. Ele sunt **strsize()**, **makestr()** şi funcţia de conversie **operator char*()** şi sunt definite în interiorul declaraţiei clasei **StrType.** Iată, mai jos, aceste funcţii:

```
int strsize() {return strlen(p);} // returneaza marimea
                                   // sirului
void makestr(char *s) {strcpy(s, p);} // formeaza siruri
                                       // intre ghilimele
operator char *() {return p;} // converteste in sir
```

Primele două funcţii sunt uşor de înţeles. După cum puteţi vedea, funcţia **strsize()** returnează lungimea unui şir spre care indică **p**. Funcţia **makestr()** copiază şirul spre care indică **p** într-o matrice de caractere. Această funcţie este folositoare când doriţi să obţineţi un şir terminat cu null, pornind de la un obiect de tip **StrType**.

Funcţia de conversie **operator char*()** returnează **p**, care este, desigur, un pointer către şirul conţinut de obiect. Această funcţie permite ca un obiect de tip **StrType** să fie folosit oriunde poate fi folosit şi un şir terminat cu null. De exemplu, acesta este un cod corect:

```
StrType x("Hello");
// afiseaza sirul folosind o functie C++ standard
puts(x); // conversie automata in char *
```

Amintiţi-vă că o funcţie de conversie este executată automat atunci când un obiect este implicat într-o expresie pentru care este definită acea conversie. În acest caz, deoarece prototipul funcţiei **puts()** cere compilatorului un argument de tip **char ***, se efectuează automat conversia din **StrType** în **char**, producând returnarea unui pointer către şirul conţinut în **x**. Datorită funcţiei de conversie, puteţi să folosiţi un obiect de tip **StrType** în locul unui şir normal, între ghilimele, ca argument pentru orice funcţie care poate prelua un argument de tip **char ***.



***NOTĂ:** Conversia în **char *** încalcă principiul încapsulării, deoarece o dată ce o funcţie are un pointer spre şirul obiectului, este capabilă să modifice direct acel şir, evitând funcţiile membre din **StrType** şi fără ca să fie implicat obiectul. De aceea, trebuie să utilizaţi cu multă grijă conversia în **char ***. (Dacă nu aveţi nevoie de ea, mai bine eliminaţi-o pur şi simplu dintre specificaţiile clasice. Pierderea încapsulării în acest caz este compensată de utilitatea şi integrarea sporită în cadrul funcţiilor de bibliotecă existente. Schimbul însă nu este întotdeauna avantajos.*

Întreaga clasă StrType

Iată listingul întregii clase **StrType** împreună cu o scurtă funcţie **main()** care îi ilustrează facilităţile:

```
#include <iostream.h>
#include <string.h>
#include <stdlib.h>
#include <conio.h>
#include <stdio.h>

class StrType {
    char *p;
    int marime;
public:
    StrType(char *sir);
    StrType();
    StrType(const StrType &0); // constructor de copie
    ~StrType() {delete [] p;}

    friend ostream &operator<<(ostream &stream, StrType &o);
    friend istream &operator>>(istream &stream, StrType &o);

    StrType operator=(StrType &o); // atribuie un obiect de
                                   // tip StrType
    StrType operator=(char *s); // atribuie un sir intre
                                // ghilimele

    StrType operator+(StrType &o); // concateneaza un
                                   // obiect de tip StrType
    StrType operator+(char *s); // concateneaza un sir
                                // intre ghilimele
    friend StrType operator+(char *s, StrType &o); /* con-
        cateneaza un sir intre ghilimele cu un obiect de
        tip StrType */

    StrType operator-(StrType &o); // exclude un subsir
    StrType operator-(char *s); // exclude un subsir intre
                                // ghilimele

    // operatii relationale intre obiecte de tip StrType
    int operator==(StrType &o) {return !strcmp(p, o.p);}
    int operator!=(StrType &o) {return strcmp(p, o.p);}
```

```
    int operator<(StrType &o) {return strcmp(p, o.p) < 0;}

    int operator>(StrType &o) {return strcmp(p, o.p) > 0;}
    int operator<=(StrType &o) {return strcmp(p, o.p) <= 0;}
    int operator>=(StrType &o) {return strcmp(p, o.p) >= 0;}

    // operatii intre obiecte de tip StrType si siruri
    // intre ghilimele
    int operator==(char *s) {return !strcmp(p, s);}
    int operator!=(char *s) {return strcmp(p, s);}
    int operator<(char *s) {return strcmp(p, s) < 0;}
    int operator>(char *s) {return strcmp(p, s) > 0;}
    int operator<=(char *s) {return strcmp(p, s) <= 0;}
    int operator>=(char *s) {return strcmp(p, s) >= 0;}

    int strsize() {return strlen(p);} // returneaza
                                      // lungimea sirului
    void makestr(char *s) {strcpy(s, p);} // creeaza siruri
                                          // intre ghilimele
    operator char *() { return p;} // conversie in char *
};

// Nici o initializare explicita
StrType::StrType() {
    marime = 1; // face loc pentru terminatia de null
    p = new char[marime];
    if(!p) {
          cout << "Eroare de alocare\n";
          exit(1);
    }
    strcpy(p, " ");
}

// Initializare folosind un sir intre ghilimele
StrType::StrType(char *str) {
    marime = strlen(sir) + 1; // face loc pentru terminatia
                              // de null
    p = new char[marime];
    if(!p) {
          cout << "Eroare de alocare\n";
          exit(1);
    }
    strcpy(p, sir);
}

// Initializare folosind un obiect de tip StrType.
StrType::StrType(const StrType &o) {
    marime = o.marime;
    p = new char[marime];
    if(!p) {
          cout << "Eroare de alocare\n;
          exit(1);
    }
    strcpy(p, o.p);
}

// Scrie un sir la iesire
ostream &operator<<(ostream &stream, StrType &o)
{
    stream << o.p;
    return stream;
}

// Introduce un sir.
istream &operator>>(istream &stream, StrType &o)
{
    char t[255]; // marime arbitrara - modificati-o daca
                 // este necesar
    int lung;

    for(lung=0; lung<255; lung++) {
          stream.da(t[lung]);
          if(t[lung]=='\n') break;
          if(t[lung]=='\b')
                if(lung) {
                      lung--;
                cout << "'\b'";
          }
    }
    t[lung] = '\0';
    lung++;

    if(lung > o.marime) {
          delete o.p;
          o.p = new char[lung];
          if(!o.p) {
```

```
            cout << "Eroare de alocare\n";
            exit(1);
        }
        o.marime = lung;
    }
    strcpy(o.p, t);
    return stream;
}

// Atribuie un obiect de tip StrType unui obiect de tip
// StrType.
StrType StrType::operator=(StrType &o)
{
    StrType temp(o.p);
    if(o.marime > marime) {
        delete p; // elibereaza memoria veche
        p = new char[o.marime];
        marime = o.marime;
        if(!p) {
            cout << "Eroare de alocare\n";
            exit(1);
        }
    }

    strcpy(p. o.p);
    strcpy(temp.p, o.p);

    return temp;
}
// Atribuie un sir intre ghilimele unui obiect de tip
// StrType.
StrType StrType::operator=(char *s)
{
    int lung = strlen(s) + 1;
    if(marime < lung) {
        delete p;
        p = new char[lung];
        marime = lung;
        if(!p) {
            cout << "Eroare de alocare\n";
            exit(1);
        }
    }
```



```
    strcpy(p, s);
    return *this;
}

// Concateneaza doua obiecte de tip StrType.
StrType StrType::operator + (StrType &o)
{
    int lung;
    StrType temp;

    delete temp.p;
    lung = strlen(o.p) + strlen(p) + 1;
    temp.p = new char[lung];
    temp.marime = lung;
    if(!temp.p) {
        cout << "Eroare de alocare\n";
        exit(1);
    }
    strcpy(temp.p, p);

    strcat(temp.p, o.p);

    return temp;
}

// Concateneaza un obiect de tip StrType cu un sir intre
// ghilimele.
StrType StrType::operator+(char *s)
{
    int lung;
    StrType temp;

    delete temp.p

    lung = strlen(s) + strlen(p) + 1;
    temp.p = new char[lung];
    temp.marime = lung;
    if(!temp.p) {
        cout << "Eroare de alocare\n";
        exit(1);
    }
    strcpy(temp.p, p);
```

```
    strcat(temp.p, s);

    return temp;
}

// Concateneaza un sir intre ghilimele cu un obiect de tip
// StrType.
StrType operator+(char *s, StrType &o)
{
    int lung;
    StrType temp;

    delete temp.p;

    lung = strlen(s) + strlen(o.p) + 1;
    temp.p = new char[lung];
    temp.marime = lung;
    if(!temp.p) {
          cout << "Eroare de alocare\n";
          exit(1);
    strcpy(temp.p, s);
    strcat(temp.p, o.p);
    return temp;
}

// Exclude un subsir dintr-un sir folosind obiecte de tip
// StrType.
StrType StrType::operator-(StrType &subsir)
{
    StrType temp(p);
    char *s1;
    int i, j;

    s1 = p;
    for(i=o; *s1; i++) {
          if(*s1!=*subsir.p) { // daca nu este prima litera
                               // a subsirului
                temp.p[i] = *s1; // atunci copiaza in temp
                s++;
          }
          else {
                for(j=0; subsir.p[j]==s1[j] && subsir.p[j];
                    j++);
```



```
                if(!subsir.p[j]) { // este subsirul, deci
                                   // trebuie eliminat
                      s1 += j;
                      i--;
                }
                else { // nu este subsirul, continua copierea
                      temp.p[i] = *s1;
                      s1++;
                }
          }
    }
    temp.p[i] = '\0';
    return temp;
}

// Exclude siruri intre ghilimele din obiecte de tip
// StrType.
StrType StrType::operator-(char *subsir)
{
    StrType temp(p);
    char *s1;
    int i, j;

    s1 = p;
    for(i=0; *s1; i++) {
          if(*s1!=*subsir) { // daca nu este prima litera
                             // din subsir
                temp.p[i] = *s1; // atunci copiaza in temp
                s1++;
          }
          else {
                for(j=0; subsir[j]==s1[j] && subsir[j];
                    j++);
                if(!subsir[j]) { // este subsirul, deci
                                 // trebuie eliminat
                      s1 += j++;
                      i--;
                }
                else { // nu este subsirul, continua copierea
                      temp.p[i] = *s1;
                      s1++;
                }
          }
```

```
    }
    temp.p[i] = '\0';
    return temp;
}

main()
{
    StrType s1("O sesiune-exemplu folosind obiecte de tip
                sir.\n");
    StrType s2(s1);
    StrType s3;
    char s[80];

    cout << s1 << s2;

    s3 = s1;
    cout << s1;

    s3.makestr(s);

    cout << "Converteste intr-un sir: " << s;

    s2 = "Acesta este un sir nou.";
    cout << s2 << endl;

    StrType s4(" Asa este.");
    s1 = s2+s4;
    cout << s1 << endl;

    if(s2==s3) cout << "Sirurile sunt egale.\n";
    if(s2!=s3) cout << "Sirurile nu sunt egale.\n";
    if(s1<s4) cout << "s1 mai mic decat s4\n";
    if(s1>s4) cout << "s1 mai mare decat s4\n";
    if(s1<=s4) cout << "s1 mai mic sau egal cu s4\n";
    if(s1>=s4) cout << "s1 mai mare sau egal cu s4\n";

    if(s2 > "ABC") cout << "s2 mai mare decat ABC\n\n";

    s1 = "unu doi trei unu doi trei\n";
    s2 = "doi";
    cout << "Sirul initial: " << s1;
    cout << "Sirul dupa excluderea lui doi: ";
    s3 = s1 - s2;
```



```
    cout << s3;

    cout << endl;
    s4 = "Va salut!";
    s3 = s4 + " sirurile din C++ sunt nostime\n";
    cout << s3;
    s3 = s3 - "Va salut!";
    s3 = "Nu-i asa ca sunt" + s3;
    cout << s3;

    s1 = s3 - "sunt ";
    cout << s1;
    s3 = s1;

    cout << "Introduceti un sir: ";
    cin >> s1;
    cout << s1 << endl;
    cout << "s1 are lungimea de " << s1.strsize()
         << " caractere.\n";

    puts(s1); // converteste in char *

    s1 = s2 = s3;
    cout << s1 << s2 << s3;

    s1 = s2 = s3 = "Pa ";
    cout << s1 << s2 << s3;

    return 0;
}
```

Precedentul program determină următoarea ieşire:

```
O sesiune-exemplu folosind obiecte de tip sir.
O sesiune-exemplu folosind obiecte de tip sir.
O sesiune-exemplu folosind obiecte de tip sir.
Converteste intr-un sir: O sesiune-exemplu folosind obiecte de tip sir.
Acesta este un sir nou.
Acesta este un sir nou. Asa este.
Sirurile nu sunt egale.
s1 este mai mare decat s4
s1 este mai mare sau egal ca s4
s2 este mai mare decat ABC
```

```
Sirul initial: unu doi trei unu doi trei
Sirul dupa excluderea lui doi: unu trei unu trei

Va salut! sirurile din C++ sunt nostime
Nu-i asa ca sunt sirurile din C++ sunt nostime
Nu-i asa ca sirurile din C++ nostime
Introduceti un sir: Imi place C++
sl are lungimea de 13 caractere.
Imi place C++
Nu-i asa ca sirurile din C++ nostime
Nu-i asa ca sirurile din C++ nostime
Nu-i asa ca sirurile din C++ nostime
Pa Pa Pa
```

Această ieşire presupune că şirul "Imi place C++" a fost scris de către utilizator la solicitarea unei introduceri.

Pentru a avea acces uşor la clasa **StrType**, îndepărtaţi funcţia **main()** şi puneţi restul listingului precedent într-un fişier numit STR.H. Apoi intoduceţi pur şi simplu acest fişier antet în orice program în care doriţi să folosiţi **StrType**.

Utilizarea clasei StrType

Drept concluzii ale acestui capitol, sunt oferite două exemple care ilustrează cum poate fi folosită clasa **StrType**. Primul exemplu creează un simplu dicţionar care foloseşte obiecte de tip **StrType**. El construieşte la început o matrice bidimensională de obiecte de acest tip. În fiecare pereche de şiruri, primul conţine cuvântul de bază după care se poate face căutarea, iar al doilea o listă de sinonime sau cuvinte înrudite. Se solicită un cuvânt, iar dacă acesta se află în dicţionar, sunt afişate sinonimele. Program este foarte simplu, dar reţineţi modul elegant în care sunt tratate şirurile, care se datorează utilizării clasei **StrType** şi operatorilor săi. (Amintiţi-vă că fişierul antet STR.H conţine clasa **StrType**.)

```
#include "str.h"
#include <iostream.h>

StrType dictionar[][2] = {
    "carte", "volum, tom",
    "magazin", "pravalie, shop, butic",
    "pistol", "arma, revolver, pistolet",
    "alergare", "fuga, jogging, cros",
    "gindire", "meditare, contemplare, reflectie",
    "calcul", "socoteala, estimare, rezolvare",
```



```
    "", ""
};

main()
{
    StrType x;

    cout << "Introduceti un cuvant: ";
    cin >> x;

    int i;
    for(i=0; dictionar[i][0]!=""; i++)
         if(dictionar[i][0]==x) cout << dictionar[i][1];
    return 0;
}
```

Următorul exemplu foloseşte un obiect de tip **StrType** pentru a afla dacă există o versiune executabilă a unui fişier pornind de la numele acestuia. Pentru a folosi programul, specificaţi în linia de comandă numele fişierului, fără nici o extensie. Programul va căuta în mod repetat un fişier executabil cu acel nume, la care adaugă o extensie, va încerca să-l deschidă şi va raporta rezultatul. (Dacă fişierul nu există, nu poate fi deschis.) După încercarea unei extensii, aceasta este exclusă din numele fişierului şi se adaugă una nouă. Din nou clasa **StrType** şi operatorii ei fac ca manevrarea şirurilor să fie uşoară şi lesne de urmărit.

```
#include "str.h"
#include <iostream.h>
#include <fstream.h>

// extensii pentru fisiere executabile
char ext[3][4] = {
    "EXE",
    "COM"'
    "BAT"
};

main(int argc, char *argv[])
{
    StrType numef;
    int i;
    if(argc!=2) {
         cout << "Utilizare: numef\n";
         return 1;
```

```
        }

        numef = argv[1];
        numef = numef + "."; // adauga un punct
        for(i=0; i<3; i++) {
              numef = numef + ext[i]; // adauga extensia
              cout << "Incearca " << numef << " ";
              ifstream f(numef);
              if(f) {
                    cout << "- Exista\n";
                    f.close();
              }
              else cout << "- Nu exista\n";
              numef = numef - ext[i]; // exclude extensia
        }
        return 0;
}
```

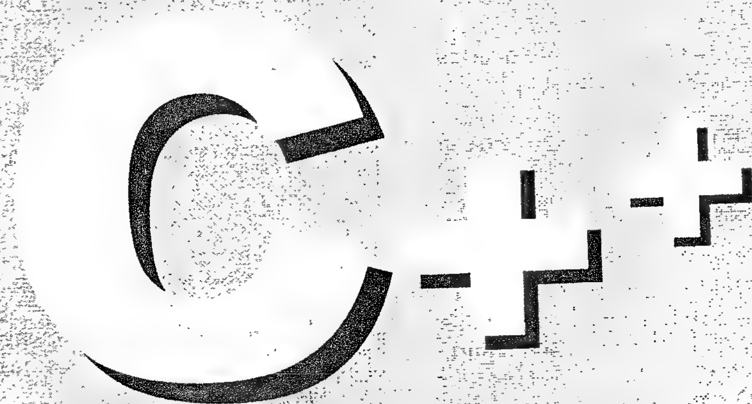
De exemplu, dacă programul de mai sus se numeşte ESTEEXEC şi presupunând că există TEST.EXE, linia de comandă **ESTEEXEC TEST** determină această ieşire:

```
Incearca TEST.EXE - Exista
Incearca TEST.BAT - Nu exista
Incearca TEST.COM - Nu exista
```

Un lucru de reţinut despre program este acela că este folosit un obiect de tip **StrType** la apelarea funcţiei **fopen()**. Operaţia funcţionează deoarece este apelată automat funcţia de conversie **operator char*()**. După cum ilustrează această situaţie, prin aplicarea atentă a facilităţilor de C++, puteţi să realizaţi o integrare semnificativă a tipurilor create de dvs. în cadrul celor standard din C++.

Capitolul 24

O clasă pentru afişarea ferestrelor



Fereastra care se deschide pe ecran a devenit una dintre cele mai cunoscute imagini în lumea calculatoarelor moderne. Este greu de închipuit astăzi scrierea unei aplicaţii comerciale care să nu folosească una sau mai multe ferestre. Modelul ferestrelor este universal. Fereastra care se deschide reprezintă, de asemenea, un exemplu clasic de programare orientată pe obiecte folosind C++. De aceea, acest capitol dezvoltă o clasă de ferestre bazate pe text şi asigură toate funcţiile necesare unui sistem de ferestre simplu, dar eficient.

De vreme ce Microsoft Windows a devenit sistemul de operare preponderent pentru PC, puteţi să vă întrebaţi dacă mai are valoare practică crearea altui sistem de ferestre. Răspunsul este da, din trei motive. Primul, sistemul creat în acest capitol se bazează pe text (nu pe grafică, aşa ca în Windows). El este proiectat special pentru a rula sub DOS sau sub emulatorul de DOS asigurat de către Windows. Este mic şi eficient şi permite multă uşurinţă în lucrul cu ferestre bazate pe text. Al doilea motiv pentru a vă crea propriul sistem de ferestre este acela că puteţi controla complet sistemul pe care îl creaţi. Deoarece echipamentul hard este în continuă dezvoltare, puteţi să vă dezvoltaţi sistemul de ferestre pentru a beneficia de avantajul unor noi moduri video sau echipamente, înainte ca facilităţile pentru acestea să devină accesibile. Puteţi, de asemenea, să îl faceţi să lucreze în noi medii şi sisteme de operare. În sfârşit, sistemul de ferestre dezvoltat în acest capitol are interfaţă directă cu echipamentul hard video pentru PC. Ca atare, el ilustrează cum C++ este capabil să intre în legătură la un nivel scăzut cu dispozitivele, cât şi cu programele de nivel înalt. Aceleaşi metode generale, care creează interfaţa pentru echipamentul video pot fi folosite şi pentru alte tipuri de dispozitive.



NOTĂ: Sistemele de ferestre sunt, în mare măsură, dependente de echipamente. Acest capitol presupune un mediu PC, iar multe funcţii operează direct cu ROM BIOS (sistemul de bază de I/O) sau chiar cu hardul. Dar, modificând doar câteva funcţii, puteţi să adaptaţi sistemul de ferestre la orice mediu.

Înainte de a începe, este important să definim ce va face un sistem de ferestre.

Ferestrele

O fereastră este o porţiune din ecran folosită pentru un anumit scop. La deschiderea ei, se salvează ceea ce se află pe ecran în mod curent şi, în locul său, se afişează fereastra. Când se încheie aplicaţia ce foloseşte fereastra, aceasta este eliminată şi este restaurat conţinutul iniţial al ecranului. Este posibil ca pe ecran să existe, în acelaşi timp, mai multe ferestre.

O caracteristică importantă a sistemului de ferestre este aceea că el nu trebuie să permită unei aplicaţii ce foloseşte fereastra să scrie peste limitele acesteia. Deoarece mărimea ferestrei nu este neapărat cunoscută aplicaţiei, rămâne în

sarcina rutinei lor pentru ferestre, şi nu a aplicaţiei, să prevină suprascrierea. De aceea, nu pot fi folosite nici rutinele normale din C++ de I/O la consolă (cum ar fi **printf()** şi **gets()**), şi nici streamurile **cin** şi **cout**; ele trebuie înlocuite cu funcţii alternative de I/O specifice ferestrelor. De fapt, aceste funcţii formează o mare parte a oricărui sistem de ferestre.

Pentru a înţelege cum pot fi folosite efectiv ferestrele, imaginaţi-vă că aţi scris un editor de texte care conţine câteva facilităţi suplimentare. Una dintre ele este un calculator de buzunar. Deoarece utilizarea sa nu face, practic, parte din editorul de texte, are sens să folosim o fereastră oricând dorim să-l folosim. Astfel, când se lucrează cu acest calculator, editarea textului este doar suspendată, nu este complet întreruptă. După efectuarea calculelor, fereastra calculatorului este închisă şi continuă editarea textului.

Crearea unor funcţii de suport video

Înainte de a construi un sistem de ferestre, este necesar să dezvoltaţi câteva funcţii de suport. Deoarece crearea ferestrelor necesită controlul direct şi eficient al ecranului, sunt necesare funcţii specializate pentru a efectua operaţii de I/O cu acesta. După cum s-a mai spus, nu este posibil să folosiţi funcţiile şi operatorii normali din C++ pentru ieşiri. Funcţiile specializate vor ocoli atât DOS cât şi BIOS şi vor scrie direct la echipamentul hardware video. Acesta este singurul mod prin care se poate realiza reactualizarea rapidă a ecranului.

Pentru început, să aruncăm o privire asupra sistemului video al calculatorului.



NOTĂ: Discuţia prezentată în acest capitol despre componenta video a calculatorului este suficientă pentru a înţelege cum lucrează sistemul de ferestre. Pentru a învăţa mai mult despre interfaţa video în general, consultaţi şi alte cărţi.

Sistemul video al calculatorului

Toate calculatoarele conţin un adaptor oarecare care produce imagini pe monitor. Cele mai întâlnite patru tipuri de adaptoare sunt adaptoarele monocrome, adaptoarele grafice/color (CGA), adaptoarele grafice îmbunătăţite (EGA) şi matricea pentru grafică video (VGA). (Mai există şi o serie de adaptoare, Super VGA, care acceptă moduri video extinse, dar, în cadrul acestui capitol, VGA şi Super VGA se vor considera echivalente.) CGA, EGA şi VGA au mai multe moduri de operare. Sistemul de ferestre dezvoltat în acest capitol cere ca sistemul video să fie în mod text cu 80 de coloane, modul implicit de operare al majorităţii aplicaţiilor bazate pe consolă ce au scopuri generale. La adaptorul monocrom, modul text cu 80 de coloane are numărul 7. La adaptoarele CGA/EGA/VGA se foloseşte modul 2 sau 3.

Caracterele afişate pe ecran sunt păstrate într-o zonă în RAM, rezervată pentru adaptoarelor de afişare. Locaţia memoriei video RAM începe la B800:0000H. Deşi CGA, EGA şi VGA funcţionează diferit în unele moduri, ele operează identic în modurile 2 şi 3.

Fiecare caracter afişat pe ecran necesită doi octeţi de memorie video. Primul octet păstrează caracterul efectiv, iar cel de-al doilea păstrează *atributul său pentru ecran*. Pentru adaptoarele color, octetul de atribut este interpretat aşa cum se arată în Tabelul 24-1. Culorile primare pot fi combinate pentru a produce culori derivate. Dacă posedaţi un CGA, EGA sau VGA, caracterele sunt afişate implicit cu octetul atribut având valoarea 7. Aceasta determină activarea pentru text a celor trei culori, care produc culoarea albă. Pentru a produce negativul video, biţii din prim-plan sunt dezactivaţi iar cei trei de fundal sunt activaţi, obţinându-se valoarea 70H.

Adaptorul monocrom recunoaşte octeţii de strălucire şi de intensitate. Din fericire, el este proiectat pentru a interpreta atributul 7 ca normal, iar 70H ca negativ video. În plus, valoarea 1 determină sublinierea caracterelor.

Fiecare adaptor are de fapt de patru ori mai multă memorie decât îi este necesară pentru a afişa un text în modul cu 80 de coloane. Motivele pentru aceasta sunt două. În primul rând, memoria suplimentară este necesară pentru grafică (desigur, cu excepţia adaptoarelor monocrome). În al doilea rând, ea permite păstrarea în RAM a mai multor ecrane pe care le schimbă, atunci când este necesar. Fiecare regiune de memorie este numită *pagină video*, iar efectul schimbării paginii video curente este de-a dreptul spectaculos. Implicit DOS şi Windows folosesc pagina 0 şi, practic, toate aplicaţiile procedează la fel. Din acest motiv, ea va fi utilizată şi în rutinele acestui capitol. Dar, puteţi, dacă doriţi, să folosiţi alte pagini.

Există trei căi de a avea acces la adaptorul video. Prima este prin apelări către sistemul de operare, o metodă care este mult prea lentă pentru ferestre. A doua este prin rutine BIOS, metodă ceva mai rapidă, ce asigură suficientă viteză la calculatoare puternice în cazul unor ferestre mici. Al treilea mod este prin citire şi scriere direct în RAM, cale care este foarte rapidă, dar necesită efort de programare mare. Totuşi, pentru ca ferestrele să apară „instantaneu" pe ecran este necesar accesul direct la memoria RAM video. Din acest motiv, cea de-a treia cale va fi folosită pentru sistemul de ferestre dezvoltat aici.

Bit	Valoare	Semnificaţie la activare
0	1	Prim-plan albastru
1	2	Prim-plan verde
2	4	Prim-plan roşu
3	8	Intensitate mică
4	16	Fundal albastru
5	32	Fundal verde
6	64	Fundal roşu
7	128	Caracter ce clipeşte

Tabelul 24-1 *Octetul pentru atributul video*

Pentru a permite ca sistemul de ferestre să aibă acces la memoria RAM video, este necesar un pointer către ea. Însă, când rulaţi un program în sistemul DOS (sau în emulatorul DOS din Windows), memoria RAM video se află într-un segment diferit de cel folosit de program. Aceasta înseamnă că trebuie utilizat un pointer FAR. Pointerii de tip FAR pot fi recunoscuţi de către un compilator de C++ într-unul din două feluri. Primul, prin cuvântul cheie suplimentar **far**, admis de multe compilatoare. (Unele compilatoare numesc acest cuvânt **_far** sau **__far**, verificaţi deci în manualul compilatorului dvs.) El permite ca un pointer să fie declarat ca FAR. A doua cale este să compilaţi programul folosind un model de memorie mare, caz în care implicit toţi pointerii sunt FAR. Rutinele folosite în acest capitol utilizează modelatorul de tip **far**. Dacă doriţi, puteţi să îl eliminaţi şi să compilaţi simplu codul folosind un model de memorie mare.

Accesul la BIOS

Chiar dacă funcţiile video care citesc şi scriu efectiv informaţii vor ocoli DOS şi BIOS şi vor avea acces direct la RAM, BIOS va mai fi folosit pentru câteva operaţii. Apelările la acesta vor fi efectuate folosind o întrerupere software. Funcţia care generează aşa ceva este **int86()**. Ea este specifică mediilor DOS şi are următorul prototip:

int int86(int *numar*, union REGS **registruintrare*, union REGS **registruiesire*);

BIOS foloseşte mai multe întreruperi diferite pentru diverse scopuri şi discutarea lor depăşeşte scopul acestui capitol. Oricum, cea legată de adaptorul video este întreruperea 16 (10H). Ca multe întreruperi pentru BIOS, întreruperea 16 are mai multe opţiuni, care sunt selectate în funcţie de valoarea registrului AH. Dacă funcţia BIOS întoarce o valoare, ea este, în general, returnată în AX. Totuşi, câteodată sunt folosite alte registre, dacă se întorc mai multe valori. Valoarea returnată de **int86()** este valoarea registrului **AX**.

Structura **REGS** este inclusă în fişierul antet DOS.H. Ea este definită astfel:

```
struct BYTEREGS {
    unsigned char al;
    unsigned char ah;
    unsigned char bl;
    unsigned char bh;
    unsigned char cl;
```

```
    unsigned char ch;
    unsigned char dl;
    unsigned char dh;
};

struct WORDREGS {
    unsigned ax;
    unsigned bx;
    unsigned cx;
    unsigned dx;
    unsigned si;
    unsigned di;
    unsigned cflag;
    unsigned flags; // Acesta nu este definit de toate
                    // versiunile de WORDREGS
};

union REGS {
    struct BYTEREGS h;
    struct WORDREGS x;
};
```

După cum puteţi vedea, **REGS** este o uniune de două structuri. Folosind structura **WORDREGS**, ea vă permite accesul la regiştrii unităţii centrale de prelucrare (CPU) ca recipiente de 16 biţi. **BYTREREGS** vă oferă acces la regiştrii individualizaţi de 8 biţi.

*NOTĂ: Microsoft C++ apelează funcţia **int86()**prin **_int86()** şi se referă la **REGS** ca **_REGS**.*

Determinarea locaţiei memoriei RAM video

Când citiţi şi scrieţi direct în memoria RAM video, mai întâi aveţi nevoie să depăşiţi problema produsă de faptul că adaptorul monocolor are memoria RAM video la B000:0000H, în timp ce celelalte îl au la B800:0000H. Pentru ca rutinele ferestrelor să opereze corect pentru fiecare adaptor, trebuie să ştie ce adaptor se află în sistem. Din fericire, există o cale simplă de a determina aceasta. Întreruperea BIOS 16, funcţia 15, returnează modul video curent. Cum s-a spus mai devreme, rutinele din acest capitol necesită modurile 2, 3 sau 7. Modurile 2 şi 3 pot fi utilizate doar de către CGA, EGA sau VGA, iar acestea nu pot folosi modul 7 - doar adaptorul monocolor o poate face. De aceea, dacă modul video curent este 7, înseamnă că este folosit un adaptor monocolor; altfel, el este de tipul CGA, EGA sau VGA. Deoarece în modul text acestea din urmă funcţionează identic, pentru sistemul de ferestre nu contează cu care dintre ele se lucrează. Astfel, folosind modul video curent, este posibil să activaţi un pointer tip FAR global spre adresa memoria RAM video. Acest lucru este efectuat de către următoarele funcţii:


```
char far *vid_mem; // pointer spre memoria ecran in modul text

void set_v_ptr()
{
    int vmod;
    vmod = mod_video();
    if((vmod!=2) && (vmod!=3) && (vmod!=7)) {
        cout << "Video trebuie sa fie in modul text cu 80
                 de coloane.";
        exit(1);
    }
    // stabileste adresa corecta a memoriei RAM video
    if(vmod==7) vid_mem = (char far *) 0xB0000000;
    else vid_mem = (char far *) 0xB8000000;
}

// Returneaza modul video curent.
mod_video()
{
    union REGS r;

    r.h.ah = 15; // da modul video
    return int86(0x10, &r, &r) & 255;
}
```

Funcţia **mod_video()** foloseşte întreruperea BIOS 16, funcţia 15, pentru a obţine modul video curent. Această valoare este folosită, apoi, pentru a determina spre ce va indica **vid_mem**, declarat ca un **char far *** global (către memoria folosită fie de adaptorul monocolor, fie de către alte adaptoare). Acest pointer va fi apoi utilizat de către rutinele bazate pe ferestre care cer acces la memoria RAM video.

Scrierea în memoria RAM video

O dată ce a fost determinat modul video şi a fost obţinut un pointer către memoria RAM video, pot fi create două funcţii de ieşire directe către aceasta. Aceste funcţii nu fac parte propriu-zis din sistemul de ferestre, dar ele sunt necesare deoarece

asigură o ieşire foarte rapidă. Funcţiile scriu un caracter sau un şir în locaţia X,Y. Este completat, de asemenea, octetul de atribut. Ele sunt prezentate aici:

```
// Scrie caracter cu atribut specificat.
void scrie_car(int x, int y, char ch, int atrib)
{
    char far *v;

    v = vid_mem;
    v += (y*160) + x*2;
    *v++ = ch; // scrie caracterul
    *v = atrib; // scrie atributul
}

// Afiseaza un sir cu atributul specificat.
void scrie_sir(int x, int y, char *p, int atrib)
{
    register int i;
    char far *v;

    v = vid_mem;
    v += (y*160) + x*2; // calculeaza adresa
    for(i=y; *p; i++) {
        *v++ = *p++; // scrie caracterul
        *v = atrib; // scrie atributul
    }
}
```

Deoarece în modul text cu 80 de coloane fiecare linie a ecranului are 80 de caractere, sunt folosiţi pentru fiecare linie 160 de octeţi (80 de octeţi pentru caractere şi 80 pentru atribute). Adresa pentru locaţia corectă în memoria RAM video este deci 160 ori coordonata Y plus de două ori coordonata X.

Poziţionarea cursorului

Când sunt efectuate operaţii de I/O folosind direct I/O din memoria RAM video, locaţia cursorului nu este actualizată automat. Aceasta înseamnă că rutinele pentru ferestre trebuie să deplaseze explicit cursorul. Următoarea funcţie efectuează această operaţie. (Funcţia nu face parte efectiv din clasa de ferestre, dar este utilizată de către aceasta. Funcţia foloseşte întreruperera BIOS 16, funcţia 2.)

```
// Trimite cursorul in pozitia X, Y specificata.
void goto_xy(int x, int y)
{
    union REGS r;

    r.h.ah = 2; // functia de adresare catre cursor
    r.h.dl = x; // coordonata coloanei
    r.h.dh = y; // coordonata rindului
    r.h.bh = 0; // pagina video
    int86(0x10, &r, &r);
}
```


Multe compilatoare asigură o funcţie „goto xy". Dacă al dvs. o are, sunteţi liber să o folosiţi în locul acesteia.

Clasa fereastră

Acum, deoarece a fost stabilită baza, poate fi dezvoltat sistemul de ferestre. Acesta este administrat de către clasa **wintype**. Iată declararea sa:

```
class wintype {
    // stabileste unde se plaseaza fereastra pe ecran
    int stgx; // coordonatele din stanga sus
    int susy;
    int drptx; // coordonatele din dreapta jos
    int josy;

    int chenar; // daca nu este 0, se afiseaza chenarul
    int activ; // nu este zero daca fereastra este afisata
            // pe ecran
    char *titlu; // mesaj pentru titlu

    int cursx, cursy; // localizarea curenta a cursorului
                      // in fereastra

    char *buf; // indica spre memoria buffer a ferestrei
    char color; // culoarea textului

    // functii particulare
    void salv_ecran(); // salveaza ecranul, astfel incat sa
                       // poata fi restaurat
    void restaur_ecran(); // restaureaza ecranul original
    void trasat_chenar(); // traseaza chenarul ferestrei
    void afisat_titlu(); // afiseaza titlul
public:
```

```
        wintype(int lx, int uy, // sus stanga
              int rx, int ly, // jos dreapta
              int b = 1, // diferit de zero pentru chenar
              char *mesaj = "" // mesaj pentru titlu
        );
  ~wintype() {winremove(); delete [] buf;}

  void winput(); // afiseaza o fereastra
  void winremove(); // sterge o fereastra
  int winputs(char *s); // scrie un sir in fereastra
  int winxy(int x, int y); // se deplaseaza la X, Y
                           // relativ la fereastra
  void wingets(char *s); // introduce un sir dintr-o
                         // fereastra
  int wingetche(); // introduce un caracter dintr-o
                   // fereastra
  void wincls(); // sterge continutul ferestrei
  void wincleol(); // sterge pana la sfarsitul liniei

  void setcolor(char c) {color = c;}
  char getcolor() {return color;}
  void setbkcolor(char c) {color = color | (c<<4);}
  char getbkcolor() {return (color>>4) & 127;}

  friend wintype &operator<<(wintype &o, char *s);
  friend wintype &operator>>(wintype &o, char *s);
};
```

Poziția ferestrei pe ecran și mărimea sa sunt determinate de variabilele **stgx**, **susy**, **drptx** și **josy**. Acestea păstrează coordonatele colțurilor din stânga sus și dreapta jos ale ferestrei.

Dacă fereastra va avea un chenar, atunci **chenar** trebuie să fie diferit de zero. Ori de câte ori fereastra se află pe ecran, **activ** este diferit de zero. Când fereastra nu este afișată, **activ** este zero. Titlul ferestrei (dacă există unul) este indicat de către **titlu**.

Localizarea curentă a cursorului în cadrul ferestrei este memorată în **cursx** și **cursy**. Toate ieșirile către fereastră sunt poziționate relativ la aceasta. Aceasta înseamnă că **cursx** și **cursy** sunt relative la fereastră, nu la ecran. Deci, dacă **cursx** este 5 iar **cursy** este 3, atunci, indiferent unde se află fereastra în cadrul ecranului, cursorul este localizat la 5, 3 *în cadrul ferestrei*. Colțul din stânga sus al ferestrei este 0, 0.

Când este afișată o fereastră, conținutul curent al ecranului este salvat în memoria indicată de **buf**. (Această memorie este alocată dinamic atunci când este

creată fereastra.) Culoarea textului este determinată de către valoarea lui **color**. Ea trebuie să fie una dintre valorile enumerate mai jos:

```
/* Culorile textului, primele 7 pot fi folosite, de
   asemenea, pentru a specifica culoarea fundalului. */
const enum clr {negru, albastru, verde, cian, rosu,
                magenta, maron, gri_deschis, gri_inchis,
                albastru_deschis, verde_deschis,
                cian_deschis, rosu_deschis,
                magenta_deschis, galben, alb,
                clipitor=128};
```

Iată funcția constructor pentru **wintype**:

```
// Construieste o fereastra.
wintype::wintype(int lx, int uy, // sus stinga
          int rx, int ly, // jos dreapta
          int b, // diferit de zero pentru chenar
          char *mesaj // mesaj pentru titlu
     )
{
   if(lx<0) lx = 0;
   if(rx>79) rx = 79;
   if(uy<0) uy = 0;
   if(ly>24) ly = 24;

   stgx = lx; susy = uy;
   drptx = rx, josy = ly;
   chenar = b;
   titlu = mesaj;
   activ = 0;
   cursx = cursy = 0;
   buf = new char[2*(drptx-stgx+1)*(josy-susy+1)];
   if(!buf) {
         cout << "Eroare de alocare.\n";
         exit(1);
   }
   color = alb;
}
```

Funcția constructor are grijă, pentru început, ca cele patru valori ale coordonatelor să fie în domeniu. Apoi inițializează datele particulare. Ea alocă, de asemenea, memoria care va fi folosită pentru a salva conținutul ecranului atunci

când se deschide fereastra. Când aceasta este dezactivată, va fi restaurat conținutul anterior al ecranului.

Rețineți că, implicit, culoarea textului este albă, cursorul este localizat la 0,0 iar fereastra nu este afişată.

*NOTĂ: Constructorul **wintype()** doar construieşte o fereastră. El nu o şi afişează. Afişarea unei ferestre este o operație separată de construirea ei.*

Destructorul **~wintype()** distruge fereastra (dacă este necesar) şi apoi eliberează memoria spre care indică **buf**.

Afişarea şi ştergerea unei ferestre

O dată construită o fereastră, ea poate fi afişată printr-o apelare a funcției **winput()**. Iată această funcție:

```
// Afiseaza o fereastra.
void wintype::winput()
{
    // activeaza fereastra
    if(!activ) { // nu este folosita in acel moment
          salv_ecran(; // salveaza ecranul curent
          activ = 1;
    }
    else return; // deja pe ecran

    if(chenar) trasat_chenar();
    afisat_titlu();

    // pozitioneaza cursorul in coltul din stanga sus
    goto_xy(stgx + cursx + 1, susy + cursy + 1);
}
```

Această funcție verifică la început dacă fereastra este deja pe ecran. În caz afirmativ, apelarea funcției **winput()** este ignorată. Altfel, conținutul curent al zonei de ecran în care va fi afişată fereastra va fi salvat prin apelarea funcției **salv_ecran()**, iar **activ** capătă valoarea 1. Dacă variabila **chenar** nu este zero, ferestrei i se desenează un chenar. Apoi este afişat titlul. În sfârşit, cursorul este plasat în fereastră. Deoarece valorile pentru **cursx** şi **cursy** sunt relative la fereastră, ele trebuie să fie adunate cu **stgx** şi respectiv cu **susy**, astfel încât cursorul să poată fi afişat în poziția corectă relativ la ecran.

În continuare este prezentată funcția **salv_ecran()**, funcție particulară din **wintype()**. Ea copiază pur şi simplu, octet cu octet, conținutul din memoria RAM video referitoare la zona în care va fi afişată fereastra în memoria spre care indică **buf**, după care şterge acea zonă a ecranului.

```
// Salveaza ecranul, el putand fi restaurat dupa
// indepartarea ferestrei.
void wintype::salv_ecran()
{
    register int i, j;
    char *buf_ptr;
    char far *v, far *t;

    buf_ptr = buf;
    v = vid_mem;
    for(i=susy; i<josy+1; i++) {
          for(j=stgx; j<drptx+1; j++) {
                t = (v + (i*160) + j*2);
                *buf_ptr++ = *t++;
                *buf_ptr++ = *t;
                *(t-1) = ' '; sterge fereastra
          }
    }
```

În continuare sunt prezentate funcțiile **trasat_chenar()** şi **afisat_titlu()**, de asemenea, funcții particulare din **wintype**.

```
// Traseaza un chenar in jurul ferestrei.
void wintype::trasat_chenar()
{
    register int i;
    char far *v, far *t;

    v = vid_mem;
    t = v;
    for(i=stgx+1; i<drptx; i++) {
          v += (susy*160) + i*2;
          *v++ = 196;
          *v = color;
          v = t;
          v += (josy*160) + i*2;
          *v++ = 196;
          *v = color;
          v = t;
    }
```

```
        for(i=susy+1; i<josy; i++) {
                v += (i*160) + stgx*2;
                *v++ = 179;
                *v = color;
                v = t;
                v += (i*160) + dreptx*2;
                *v++ = 179;
                *v = color;
                v = t;
        }
        // traseaza colturile
        scrie_car(stgx, susy, 218, color);
        scrie_car(stgx, josy, 192, color);
        scrie_car(drptx, susy, 191, color);
        scrie_car(drptx, josy, 217, color);
}

// Afiseaza titlul ferestrei.
void wintype::afisat_titlu()
{
        register int x, lung;

        x = stgx;

        /* Calculeaza pozitia de pornire corecta pentru a centra
           titlul - daca este negativ, mesajul nu se va incadra.
        */
        lung = strlen(titlu);
        lung = (drptx - x - lung) / 2;
        if(lung<0) return; // nu il afiseaza
        x = x + lung + 1;
        scrie_sir(x, susy, titlu, color);
}
```

Valorile 196 şi 179 din setul de caractere extins al calculatorului corespund liniilor orizontale şi verticale. Valorile folosite la sfârşitul funcţiei **trasat_chenar()** sunt caracterele pentru colţuri. Titlul este afişat doar dacă are loc în fereastră.

Pentru a îndepărta o fereastră de pe ecran, folosiţi funcţia **winremove()**, prezentată în continuare. Ea copiază înapoi în memoria RAM video ceea ce este păstrat în memoria spre care indică **buf**.

```
// Indeparteaza fereastra si restaureaza continutul
// initial al ecranului.
```



```
void wintype::winremove()
{
        if(!activ) return; // nu poate sa indeparteze o
                           // fereastra inactiva
        restaur_ecran(); // restaureaza ecranul initial
        activ = 0; // restaurare video
}
```

În afară de a da variabilei **activ valoarea 0**, scopul principal al funcţiei este de a restaura conţinutul iniţial al ecranului prin apelarea funcţiei particulare **restaur_ecran()**, aşa cum se arată mai jos:

```
// Restaureaza o zona a ecranului.
void wintype::restaur_ecran()
{
        register int i, j;
        char far *v, far *t;
        char *buf_ptr;

        buf_ptr = buf;
        v = vid_mem;
        t = v;
        for(i=susy; i<josy+1; i++)
                for(j=stgx; j<drptx+1; j++) {
                        v = t;
                        v += (i*160) + j*2;
                        *v++ = *buf_ptr++; // scrie caracterul
                        *v = *buf_ptr++; // scrie atributul
                }
}
```

I/O pentru ferestre

Intrarea din şi ieşirea într-o fereastră pot fi efectuate ori cu funcţiile membre ori cu operatorii supraîncărcaţi << şi >>. Toate operaţiile de I/O pentru ferestre trebuie să fie efectuate prin rutine care să împiedice depăşirea limitelor acestora.

Funcţia de intrare de nivel scăzut se numeşte **wingetche()**. Ea citeşte un caracter de la tastatură şi îl scrie în ecou în fereastră, aşa cum se arată aici:

```
/* Introduce o tasta apasata in interiorul unei ferestre.
   Returneaza codul complet pe 16 biti al tastei.
*/
```

```
int wintype::wingetche()
{
    union inkey {
          char ch[2];
          int i;
    } c;
    union REGS r;
    if(!activ) return 0; // fereastra nu este activa

    winxy(cursx, cursy);

    r.h.ah = 0; // citeste o tasta
    c.i = int86(0x16, &r, &r);

    if(c.ch[0]) {
          switch(c.ch[0]) {
                case '\r': // este apasata tasta ENTER
                      break;
                case '\b': // back space
                      break;
                default:
                      if(cursx + stgx < drptx - 1) {
                            scrie_char(stgx+ cursx + 1,
                            susy + cursy + 1, c.ch[0], color);
                            cursx++;
                      }
          }
          if(cursy < 0) cursy = 0;
          if(cursy + susy > josy - 2)
                cursy--;
          winxy(cursx, cursy);
    }
    return c.i;
}
```

Această funcţie apelează întreruperea BIOS 16, funcţia 0, care aşteaptă o apăsare de tastă şi returnează codul complet pe 16 biţi al tastei. Acesta este împărţit în două: caracterul şi codul de poziţie. Dacă tasta apăsată este o tastă caracter, acesta este returnat în cei 8 biţi de ordin inferior. Dacă însă este apăsată o tastă specială, pentru care nu există cod de caracter, cum ar fi o săgeată, atunci octetul de ordin inferior este zero, iar cel de ordin superior conţine codul de poziţie al tastei. De exemplu, codul de poziţie al săgeţilor în sus şi în jos este 72 şi, respectiv, 80. Deşi nici o funcţie pentru ferestre nu utilizează acest cod, el este

prevăzut aici deoarece foarte probabil aplicaţiile dvs. cu ferestre vor avea nevoie de acces atât la caractere cât şi la coduri de poziţie.

După cum puteţi vedea examinând funcţia, nici o tastare nu va avea ecou dincolo de limitele ferestrei. De asemenea, caracterele se afişează în culoarea definită curent pentru fereastră. În sfârşit, fereastra trebuie să fie activă.

Pentru a citi un şir de la tastatură, folosiţi **wingets()**, prezentat aici:

```
// Citeste un sir dintr-o fereastra.
void wintype::wingets(char *s)
{
   char ch, *temp;

   temp = s;
   for(;;) {
    ch = wingetche();
    switch(ch) {
      case '\r': // este apasata tasta ENTER
         *s = '\0';
         return;
      case '\b': // backspace
         if(s>temp) {
              s--;
              cursx--;
              if(cursx < 0) cursx = 0;
              winxy(cursx, cursy);
              scrie_char(stg x + cursx + 1, susy + cursy +
                         1, ' ', color);
         }
         break;
      default: *s = ch;
         s++;

    }
}
```

Aceasta funcţie apelează **wingetche()** pentru a introduce fiecare caracter. Deoarece **wingetche()** împiedică afişările să treacă de limitele unei ferestre, acestea nu pot fi depăşite nici de vreo intrare din **wingets()**.

Pentru a obţine la ieşire un şir într-o fereastră, folosiţi **winputs()**, prezentată aici:

```
/* Scrie un sir in pozitia curenta a cursorului
   in fereastra specificata.
```

```
    Returneaza 0 daca fereastra nu este activa;
    altfel, returneaza 1.
*/
int wintype::winputs(char *s)
{
    register int x, y;
    char far *v;

    // se asigura ca fereastra este activa
    if(!activ) return 0;
    x = cursx + stgx + 1;
    y = cursy + susy + 1;

    v = vid_mem;
    v += (y*160) + x*2; // calculeaza adresa de pornire

    for( ; *s; s++) {
        if(y >= josy) {
            return 1;
        }
        if(x >= drptx) {
            return 1;
        }

        if(*s=='\n') {
            y++;
            x = stgx + 1;
            v = vid_mem;
            v += (y*160) + x*2; // calculeaza adresa
            cursy++; // incrementeaza Y
            cursx = 0; // aduce pe x la 0;
        }
        else {
            cursx++;
            x++;
            *v++ = *s; // scrie caracterul
            *v++ = color; // culoare
        }
        winxy(cursx, cursy);
    }
    return 1;
}
```


Această funcţie afişează şirul specificat începând de la poziţionarea cursorului (dată de **cursx** şi de **cursy**), în culoarea curentă. Nu va fi permisă nici o ieşire dincolo de limitele ferestrei.

După cum s-a menţionat, în afară de funcţiile membre de I/O, puteţi, de asemenea, să afişaţi un şir folosind << şi să introduceţi unul cu >>. În continuare sunt prezentaţi aceşti operatori supraîncărcaţi:

```
// Iesire intr-o fereastra.
wintype &operator<<(wintype &o, char *s)
{
    o.winputs(s);
    return o;
}
// Intrare de la o fereastra.
wintype &operator>>(wintype &o, char *s)
{
    o.wingets(s);
    return o;
}
```

Dacă doriţi să introduceţi alt tip de date, supraîncărcaţi din nou << şi >>.

Alte trei funcţii de ieşire pentru ferestre sunt **wincls()**, care şterge conţinutul ferestrei, **wincleol()**, care şterge de la poziţia curentă a cursorului până la sfârşitul liniei, şi **winxy()**, care plasează cursorul în poziţia X, Y relativă la fereastră. Iată aceste funcţii:

```
// Sterge continutul unei ferestre.
void wintype::wincls()
{
    register int i, j;
    char far *v, far *t;

    v = vid_mem;
    t = v;
    for(i=susy+1; i<josy; i++)
        for(j=stgx+1; j<drptx; j++ {
            v = t;
            v += (i*160) + j*2;
            *v++ = ' '; // scrie un spatiu
            *v = color; // in culoarea fundalului
        }
    cursx = 0;
    cursy = 0;
```

```
}

// Sterge pana la sfarsitul liniei.
void wintype::wincleol()
{
    register int i, x, y;

    x = cursx;
    y = cursy;
    winxy(cursx, cursy);

    for(i=cursx; i<drptx-1; i++)
        winputs(" ");
    winxy(x, y);
}

/* Plaseaza cursorul intr-o fereastra intr-o pozitie
   specificata. Returneaza 0 daca este in afara limitelor;
   altfel, diferit de zero.
*/
int wintype::winxy(int x, int y)
{
    if(x<0 || x+stgx >= right-1)
        return 0;
    if(y<0 || y+susy >= josy-1)
        return 0;
    cursx = x;
    cursy = y;
    goto_xy(leftx+x+1, susy+y+1);
    return 1;
}
```

Puteţi să stabiliţi culoarea de prim-plan folosind **setcolor()**, iar cea de fundal cu **setbkcolor()**. Aceste funcţii sunt definite în interiorul declarării clasei **wintype**. Pentru **setcolor()** puteţi folosi orice culoare specificată în enumerarea **clr**. Pentru **setcolor()** puteţi să utilizaţi primele şapte culori. Funcţiile **getbkcolor()** şi **setbkcolor()** puteţi să utilizaţi primele şapte culori. Funcţiile **getbkcolor()** şi **getcolor()** returnează culorile pentru fundal şi, respectiv, pentru prim-plan. Ele sunt definite inline în interiorul clasei **wintype**.

Întregul sistem de ferestre

Iată întregul sistem de ferestre, plus funcţiile de suport şi o funcţie **main()** care ilustrează funcţiile pentru ferestre:


```
// O clasa de ferestre.

#include <iostream.h>
#include <conio.h>
#include <stdlib.h>
#include <string.h>
#include <dos.h>
#include <bios.h>

// Functii globale
int mod_video();
void goto_xy(int x, int y);
void set_v_ptr();
void scrie_car(int x, int y, char ch, int attrib);
void scrie_sir(int x, int y, char *p, int attrib);

/* Culorile textului, primele 7 pot fi folosite, de
   asemenea, pentru a specifica culoarea fundalului. */
const enum clr {negru, albastru, verde, cian, rosu,
               magenta, maron, gri_deschis, gri_inchis,
               albastru_deschis, verde_deschis,
               ciano_deschis, rosu_deschis,
               magenta_deschis, galben, alb,
               clipitor=128};

char far *vid_mem; // pointer spre memoria ecranului in
                   // modul text

class wintype {
    // stabileste unde se plaseaza fereastra pe ecran
    int stgx; // coordonatele din stanga sus
    int susy;
    int drptx; // coordonatele din dreapta jos
    int josy;

    int chenar; // daca nu este 0, se afiseaza chenarul
    int activ; // nu este zero daca fereastra este afisata
               // pe ecran
    char *titlu; // mesaj pentru titlu

    int cursx, cursy; // localizarea curenta a cursorului
                      // in fereastra
```

```
    char *buf; // indica spre memoria buffer a ferestrei

    char color; // culoarea textului

    // functii particulare
    void salv_ecran(); // salveaza ecranul, astfel incat sa
                       // poata fi restaurat
    void restaur_ecran(); // restaureaza ecranul original
    void trasat_chenar(); // traseaza chenarul ferestrei
    void afisat_titlu(); // afiseaza titlul
    public:
          wintype(int lx, int uy, // sus stanga
                int rx, int ly, // jos dreapta
                int b = 1, // diferit de zero pentru chenar
                char *mesaj = "" // mesaj pentru titlu
    );

    ~wintype() {winremove(); delete [] buf;}

    void winput(); // afiseaza o fereastra
    void winremove(); // sterge o fereastra
    int winputs(char *s); // scrie un sir in fereastra
    int winxy(int x, int y); // se deplaseaza la X, Y
                             // relativ la fereastra
    void wingets(char *s); // introduce un sir dintr-o
                           // fereastra
    int wingetche(); // introduce un caracter dintr-o
                     // fereastra
    void wincls(); // sterge continutul ferestrei
    void wincleol(); sterge pana la sfarsitul liniei

    void setcolor(char c) {color = c;}
    char getcolor() {return color;}
    void setbkcolor(char c) {color = color | (c<<4);}
    char getbkcolor() {return (color>>4) & 127;}

    friend wintype &operator<<(wintype &o, char *s);
    friend wintype &operator>>(wintype &o, char *s);
};

// Construieste o fereastra.
wintype::wintype(int lx, int uy, // sus stinga
        int rx, int ly, // jos dreapta
```



```
        int b, // diferit de zero pentru chenar
        char *mesaj // mesaj pentru titlu
    )
{
    if(lx<0) lx = 0;
    if(rx>79) rx = 79;
    if(uy<0) uy = 0;
    if(ly>24) ly = 24;
    stgx = lx; susy = uy;
    drptx = rx, josy = ly;
    chenar = b;
    titlu = mesaj;
    activ = 0;
    cursx = cursy = 0;
    buf = new char[2*(drptx-stgx+1)*(josy-susy+1)];
    if(!buf) {
          cout << "Eroare de alocare.\n";
          exit(1);
    }
    color = alb;
}

// Afiseaza o fereastra
void wintype::winput()
{
    // activeaza fereastra
    if(!activ) { // nu este folosita in acel moment
          salv_ecran(); // salveaza ecranul curent
          activ = 1;
    }
    else return; // deja pe ecran

    if(chenar) trasat_chenar();
    afisat_titlu();

    // pozitioneaza cursorul in coltul din stanga sus
    goto_xy(stgx + cursx + 1, susy + cursy + 1);
}

// Indeparteaza fereastra si restaureaza continutul initial
// al ecranului.
void wintype::winremove()
{
```

```
    if(!activ) return; // nu poate sa indeparteze o
                       // fereastra inactiva
    restaur_ecran(); // restaureaza ecranul initial
    activ = 0; // restaurare video
}

// Traseaza un chenar in jurul ferestrei.
void wintype::trasat_chenar()
{
    register int i;
    char far *v, far *t;

    v = vid_mem;
    t = v;
    for(i=stgx+1; i<drptx; i++) {
          v += (susy*160) + i*2;
          *v++ = 196; *v = color;
          v = t;
          v += (josy*160) + i*2;
          *v++ = 196;
          *v = color;
          v = t;
    }
    for(i=susy+1; i<josy; i++) {
          v += (i*160) + stgx*2;
          *v++ = 179;
          *v = color;
          v = t;
          v += (i*160) + drptx*2;
          *v++ = 179;
          *v = color;
          v = t;
    }

    // traseaza colturile
    scrie_car(stgx, susy, 218, color);
    scrie_car(stgx, josy, 192, color);
    scrie_car(drptx, susy, 191, color);
    scrie_car(drptx, josy, 217, color);
}

// Afiseaza titlul ferestrei.
void wintype::afisat_titlu()
```



```
{
    register int x, lung;

    x = stgx;

    /* Calculeaza pozitia de pornire corecta pentru a centra
    titlul - daca este negativ, mesajul nu se va incadra.
    */
    lung = strlen(titlu);
    lung = (drptx - x - lung) / 2;
    if(lung<0) return; // nu il afiseaza
    x = x + lung + 1;
    scrie_sir(x, susy, titlu, color);
}

// Salveaza ecranul, el putand fi restaurat dupa
// indepartarea ferestrei.
void wintype::salv_ecran()
{
    register int i, j;
    char *buf_ptr;
    char far *v, far *t;

    buf_ptr = buf;
    v = vid_mem;
    for(i=susy; i<josy+1; i++) {
          for(j=stgx; j<drptx+1; j++) {
                t = (v + (i*160) + j*2);
                *buf_ptr++ = *t++;
                *buf_ptr++ = *t;
                *(t-1) = ' '; // sterge fereastra
    }
}

// Restaureaza o zona a ecranului.
void wintype::restaur_ecran()
{
    register int i, j;
    char far *v, far *t;
    char *buf_ptr;

    buf_ptr = buf;
    v = vid_mem;
```

```
    t = v;
    for(i=susy; i<josy+1; i++)
          for(j=stgx; j<drptx+1; j++) {
                v = t;
                v += (i*160) + j*2;
                *v++ = *buf_ptr++; // scrie caracterul
                *v = *buf_ptr++; // scrie atributul
    }
}

/* Scrie un sir in pozitia curenta a cursorului
   in fereastra specificata.
   Returneaza 0 daca fereastra nu este activa;
   altfel, returneaza 1.
*/
int wintype::winputs(char *s)
{
    register int x, y;
    char far *v;

    // se asigura ca fereastra este activa
    if(!activ) return 0;

    x = cursx + stgx + 1;
    y = cursy + susy + 1;

    v = vid_mem;
    v += (y*160) + x*2; // calculeaza adresa de pornire
    for( ; *s; s++) {
          if(y >= josy) {
                return 1;
          }

          if(x >= drptx) {
                return 1;
          }
          if(*s=='\n') {
                y++;
                x = stgx + 1;
                v = vid_mem;
                v += (y*160) + x*2; // calculeaza adresa
                cursy++; // incrementeaza Y
                cursx = 0; // aduce pe x la 0;
```



```
          }
          else {
                cursx++;
                x++;
                *v++ = *s; // scrie caracterul
                *v++ = color; // culoare
          }
          winxy(cursx, cursy);
    }
    return 1;
}

/* Plaseaza cursorul intr-o fereastra intr-o pozitie
   specificata. Returneaza 0 daca este in afara limitelor;
   altfel, diferit de zero.
*/
int wintype::winxy(int x, int y)
{
    if(x<0 || x+stgx >= right-1)
          return 0;
    if(y<0 || y+susy >= josy-1)
          return 0;
    cursx = x;
    cursy = y;
    goto_xy(left+x+1, susy+y+1);
    return 1;
}

// Citeste un sir dintr-o fereastra.
void wintype::wingets(char *s)
{
    char ch, *temp;

    temp = s;
    for(;;) {
     ch = wingetche();
     switch(ch) {
        case '\r': // este apasata tasta ENTER
           *s = '\0';
           return;
        case '\b': // backspace
           if(s > temp) {
                s--;
```

```
                cursx--;
                if(cursx < 0) cursx=0;
                winxy(cursx, cursy);
                scrie_car(stgx + cursx + 1, susy + cursy +
                        1, ' ', color);
            }
            break;
        default: *s = ch;
            s++;
        }
    }
}
/* Introduce o tasta apasata in interiorul unei ferestre.
   Returneaza codul complet pe 16 biti al tastei.
*/
wintype::wingetche()
{
    union inkey {
        char ch[2];
        int i;
    } c;
    union REGS r;
    if(!activ) return 0; // fereastra nu este activa

    winxy(cursx, cursy);

    r.h.ah = 0; // citeste o tasta
    c.i = int86(0x16, &r, &r);

    if(c.ch[0]); {
        switch(c.ch[0]) {
            case '\r': // este apasata tasta ENTER
                break;
            case '\b': // back space
                break;
            default:
                if(cursx + stgx < drptx - 1) {
                    scrie_car(stgx + cursx + 1,
                    susy + cursy + 1, c.ch[0], color);
                    cursx++;
                }
        }
        if(cursy < 0) cursy = 0;
```



```
        if(cursy + susy > josy - 2)
            cursy--;
        winxy(cursx, cursy);
    }
    return c.i;
}

// Sterge continutul ferstrei.
void wintype::wincls()
{
    register int i, j;
    char far *v, far *t;

    v = vid_mem;
    t = v;
    for(i=susy+1; i<josy; i++)
        for(j=stgx+1; j<drptx; j++ {
            v = t;
            v += (i*160) + j*2;
            *v++ = ' '; // scrie un spatiu
            *v = color; // in culoarea fundalului
        }
    cursx = 0;
    cursy = 0;
}

// Sterge pana la sfarsitul liniei.
void wintype::wincleol()
{
    register int i, x, y;

    x = cursx;
    y = cursy;
    winxy(cursx, cursy);

    for(i=cursx; i<drptx-1; i++)
        winputs(" ");
    winxy(x, y);
}
// Iesire intr-o fereastra.
wintype &operator<<(wintype &o, char *s)
{
    o.winputs(s);
```

```
    return o;
}

// Intrare de la o fereastra.
wintype &operator>>(wintype &o, char *s)
{
    o.wingets(s);
    return o;
}

main()
{
    char s[80];

    set_v_ptr(); // da pointerul pentru memoria video

    wintype w1(1, 10, 20, 20, 1, "Fereastra Mea #1");
    wintype w2(40, 1, 60, 20, 1, "Fereastra mea #2");
    wintype w3(40, 5, 60, 20, 1, "Fereastra mea #3");

    w1.winput();
    w2.winput();
    w1.setcolor(rosu);
    w2.setcolor(verde);
    w1 >> s;
    w1.winxy(0, 0);
    w1.winputs("Va salut\n");
    w1.winputs("Ferestrele sunt distractive");
    w1 << "\n";
    w1 >> s;
    w1 << "Acesta \neste " << "un test" << "\n";
    w2 << "acesta este un test";
    w2.winxy(3, 4);
    w2 << "la locatia 3, 4";
    w2 << "Acesta este un alt test pe care sa il vedeti\n";
    w1 << "Acesta este un alt test pe care sa il vedeti\n";
    w1 >> s;
    w3.winput(); // se suprapune peste alta fereastra
    w3 >> s;
    w3.winremove();
    w1.winxy(0, 0);
    w1.setcolor (rosu);
    w1.setbkcolor (cian);
    w1.winputs("PROMPT: ");
```



```
    w1.setcolor (negru);
    w1 >>s;
    w2.winxy(0, 4);
    w2.setcolor (galben);
    w2.setbkcolor (verde);
    w2.winputs(s);
    w2 >> s;
    w1.wincls();
    w1.winxy(5, 0);
    w1.wincleol();
    w2 >> s;
    w1.winremove();
    w2.winremove();
    w1.winput ();
    w1 >> s;
    return 0;
}

void set_v_ptr()
{
    int vmod;

    vmod = mod_video();
    if((vmod!=2) && (vmod!=3) && (vmod!=7)) {
          cout << "Video trebuie sa fie in modul text cu 80
                    de coloane. ";
          exit(1);
    }
    // stabileste adresa corecta a memoriei RAM video
    if(vmod==7) vid_mem = (char far *) 0xB0000000;
    else vid_mem = (char far *) 0xB8000000;
}

// Returneaza modul video curent.
mod_video()
{
    union REGS r;

    r.h.ah = 15; // da modul video
    return int86(0x10, &r, &r) & 255;
}

// Scrie caracter cu atribut specificat.
```

```
void scrie_car(int x, int y, char ch, int atrib)
{
    char far *v;

    v = vid_mem;
    v += (y*160) + x*2;
    *v++ = ch; // scrie caracterul
    *v = atrib; // scrie atributul
}

// Trimite cursorul in pozitia X,Y specificata.
void goto_xy(int x, int y)
{
    union REGS r;

    r.h.ah = 2; // functia de adresare catre cursor
    r.h.dl = x; // coordonata coloanei
    r.h.dh = y; // coordonata randului
    r.h.bh = 0; // pagina video
    int86(0x10, &r, &r);
}

// Afiseaza un sir cu atribut specificat.
void scrie_sir(int x, int y, char *p, int atrib)
{
    register int i;
    char far *v;

    v = vid_mem;
    v += (y*160) + x*2; // calculeaza adresa
    for(i=y; *p; i++) {
        *v++ = *p++; // scrie caracterul
        *v++ = atrib; // scrie atributul
    }
}
```

Acest program produce ieşirea prezentată în **Figura 24-1**.

Lucruri de încercat

Chiar dacă sistemul de ferestre este complet funcţional, ar fi unele modificări pe care aţi putea să le încercaţi. Pentru moment, când o fereastră este închisă, conţinutul ei se pierde. Aţi putea să-i salvaţi conţinutul, astfel încât să poată fi

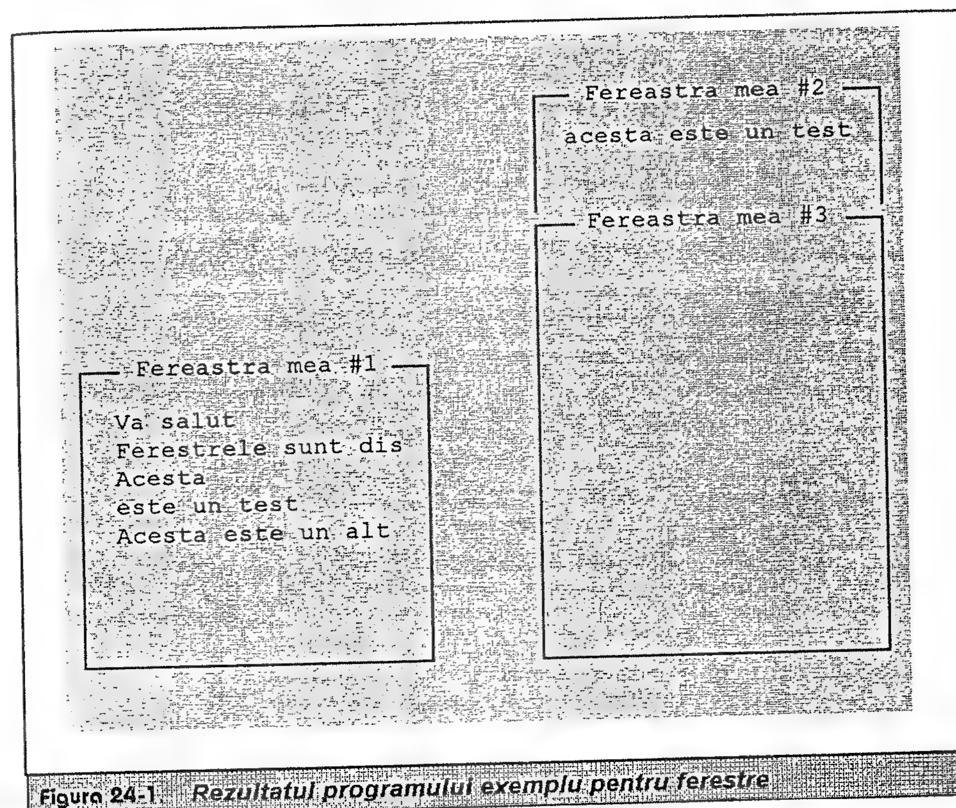


Figura 24-1. *Rezultatul programului exemplu pentru ferestre*

restaurat atunci când se reafişează fereastra. Această modificare poate fi de mare folos în anumite situaţii. Puteţi să supraîncărcaţi operatorii << şi >>, astfel încât să poată lucra şi cu alte tipuri de date, nu numai cu şiruri. Puteţi supraîncărca operatorul = relativ la obiecte de tipul **wintype**. Dacă o faceţi, asiguraţi-vă că, pentru a păstra conţinutul unei ferestre, fiecare obiect foloseşte propria sa memorie. În sfârşit, dacă doriţi, puteţi utiliza un obiect de tip fereastră pentru a iniţializa un altul, va trebui însă să creaţi un constructor de copie care să determine ca fiecărui obiect să i se aloce o memorie proprie. Dacă nu o faceţi, atunci când se distrug obiectele va fi eliberată de două ori aceeaşi zonă de memorie.

Capitolul 25

O clasă generică de liste înlănţuite



Ultimul capitol al acestei cărţi examinează problemele care apar când creaţi o clasă generică de liste dublu înlănţuite. Clasele generice sunt unele dintre cele mai importante caracteristici din C++ - în special în mediile de programare profesionale. Chiar dacă o listă dublu înlănţuită este doar una dintre metodele multiple folosite pentru stocarea informaţiilor, problemele şi soluţiile asociate creării unei clase generice de acest tip pot fi generalizate pentru oricare metodă de memorare.



*REŢINEŢI: O clasă generică este una formată, folosind cuvântul cheie **template**. Tipul de date asupra căruia operează este specificat ca parametru la iniţializarea fiecărui obiect al clasei.*

Crearea unei clase generice de liste înlănţuite necesită folosirea unor caracteristici de C++ dintre cele mai avansate şi mai abstracte. Din acest motiv, capitolul va începe cu introducerea unei clase de liste duble înlănţuite care nu este generică. Ea creează o listă dublu înlănţuită pentru un anume tip de date, indicat explicit în program. Această versiune particulară a clasei de liste înlănţuite este folosită pentru a dezvolta, a prezenta şi a explica mecanismul de bază al listelor înlănţuite. Apoi, ea va fi transformată într-una generică, ce poate lucra cu orice tip de date.

O clasă simplă de liste dublu înlănţuite

După cum ştiţi, probabil, listele dublu înlănţuite sunt structuri de date dinamice care pot să îşi mărească sau să îşi micşoreze lungimea în timpul execuţiei programului. De fapt, principalul avantaj al unei structuri de date dinamice este acela că mărimea sa nu trebuie să fie fixată în timpul compilării, ci este liberă să se dezvolte sau să se comprime după necesităţi în timpul rulării. Fiecare obiect din listă conţine o legătură către obiectul precedent şi una către următorul. Obiectele sunt introduse şi distruse din listă prin modificarea corespunzătoare a legăturilor. Deoarece listele dublu înlănţuite sunt structuri de date dinamice, de obicei fiecărui obiect memoria îi este alocată dinamic. Acesta este cazul şi cu clasele de liste dublu înlănţuite dezvoltate în acest capitol.

Fiecare element memorat într-o listă dublu înlănţuită conţine trei părţi: un pointer spre următorul element din listă, unul spre elementul precedent şi informaţiile conţinute în listă. **Figura 25-1** prezintă o astfel de listă. Listele pot memora orice tip de date, inclusiv caractere, întregi, structuri, clase, uniuni ş.a.m.d. Clasa de liste dublu înlănţuite dezvoltată în acest paragraf memorează doar caractere (pentru uşurinţa ilustrării), dar ar fi putut fi folosit orice alt tip de date.

Lista dublu înlănţuită este introdusă folosind o simplă ierarhie de clase. O clasă numită **dbinlob** defineşte natura obiectelor care vor fi memorate în listă. Această

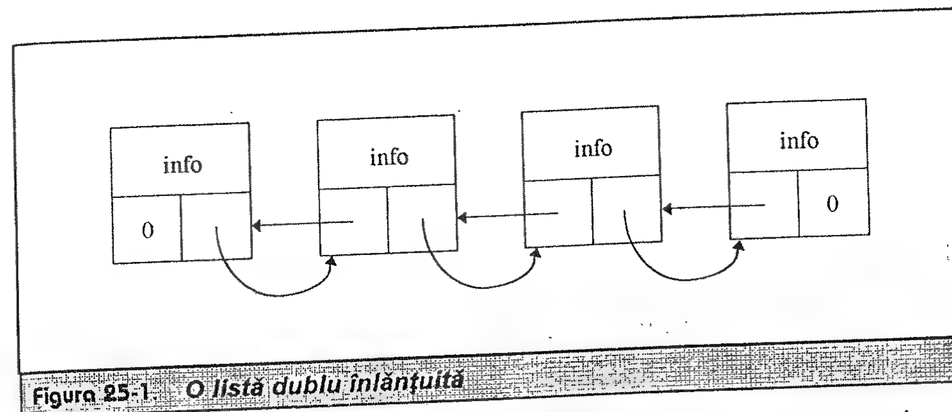


Figura 25-1. O listă dublu înlănţuită

clasă este apoi moştenită de alta, numită **dllist**, care introduce efectiv mecanismul listei dublu înlănţuite.

Clasa prezentată aici, **dbinlob**, defineşte natura fiecărui element din listă:

```
// Aceasta clasa defineste fiecare element din lista.
class dbinlob {
public:
    char info; // informatii
    dbinlob *urmator; // pointer spre obiectul urmator
    dbinlob *anterior; // pointer spre obiectul anterior
    dbinlob() {
          info = 0;
          urmator = NULL;
          anterior = NULL;
    }
    dbinlob(char c) {
          info = c;
          urmator = NULL;
          anterior = NULL;
    }
    dbinlob *daurmator() {return urmator;}
    dbinlob *daanterior() {return anterior;}
    void dainfo(char &c) { c = info;}
    void schimba(char c) { info = c; } // modifica un element

    // Supraincarca << pentru obiecte de tip dbinlob.
    friend ostream &operator<<(ostream &stream, dbinlob o)
    {
          stream << o.info << "\n";
          return stream;
```

```
    }

    // Supraincarca << pentru pointeri spre obiecte
    // de tip dbinlob.
    friend ostream &operator<<(ostream &stream, dbinlob *o)
    {
        stream << o->info << "\n";
        return stream;
    }

    // Supraincarca >> pentru referinte dbinlob.
    friend istream &operator>>(istream &stream, dbinlob &o)
    {
        cout << "Introduceti informatile: ";
        stream >> o.info;
        return stream;
    }
};
```

După cum puteţi vedea, **dbinlob** are trei membri de tip date. Membrul **info** păstrează informaţia memorată de către listă. Amintiţi-vă deocamdată tipul datelor este menţionat explicit în program drept **char**. De aceea, lista înlănţuită va fi capabilă să păstreze doar caractere. Pointerul **urmator** va indica spre următorul element al listei, iar **anterior** către elementul precedent. Reţineţi că membrii de tip date din **dbinlob** sunt publici. Ei sunt declaraţi astfel doar pentru prezentare şi pentru a permite ilustrarea mai uşoară a tuturor aspectelor listelor înlănţuite. În aplicaţiile dvs. puteţi să îi stabiliţi particulari sau protejaţi.

În **dbinlob** sunt definite, de asemenea, un număr de operaţii care pot fi efectuate asupra obiectelor de tip **dbinlob**. Anume, informaţiile asociate unui obiect pot fi extrase sau modificate şi pot fi obţinuţi pointerii spre elementul precedent şi următor. De asemenea, obiectele de tip **dbinlob** pot fi introduse şi obţinute la ieşire folosind operatorii suprapuşi << şi >>. Reţineţi că operaţiile definite în cadrul clasei **dbinlob** sunt independente de mecanismul de păstrare a listelor. **dbinlob** defineşte doar natura datelor care trebuie să fie memorate în listă.

Când este construit fiecare obiect, câmpurile **anterior** şi **următor** sunt iniţializate cu **NULL**. Pointerii rămân astfel până când obiectul este introdus într-o listă. Dacă este prezentă o valoare de iniţializare, ea este copiată în **info**, altfel **acesta** este iniţializat cu zero.

Funcţia **daurmator()** returnează un pointer spre elementul următor din listă. Dacă s-a ajuns la sfârşitul listei, acesta va fi **NULL**. Funcţia **daanterior()** întoarce un pointer spre elementul anterior din listă, dacă acesta există; altfel ea returnează **NULL**. Funcţiile nu sunt, practic, necesare deoarece **urmator** şi **anterior** sunt publice; veţi avea însă nevoie de ele când în aplicaţiile dvs. pointerii vor fi particulari.

Observaţi că operatorul << este supraîncărcat atât pentru obiecte de tip **dbinlob** cât şi pentru pointeri spre obiecte de tip **dbinlob**. Deoarece este uzual ca, atunci când folosiţi o listă înlănţuită, să aveţi acces la membrii listei folosind pointeri, este necesar să supraîncărcaţi << astfel încât să funcţioneze şi atunci când se transmite un pointer către obiect. Însă, deoarece nu există nici un motiv să interziceţi afişarea directă a unui obiect, a fost introdusă şi cea de-a doua formă, care operează direct asupra unui obiect.

Deşi **dbinlob** defineşte natura unei liste de obiecte înlănţuite, nu ea este cea care le creează. Mecanismul de înlănţuire a listelor este furnizat de clasa **dllist**, prezentată mai jos. După cum puteţi vedea, ea moşteneşte **dbinlob** şi operează asupra obiectelor de acest tip.

```
// Aceasta clasa introduce, de fapt, lista dublu inlantuita.
class dllist : public dbinlob {
    dbinlob *incep, *sfirs;
public:
    dllist() { incep = sfirs = NULL; }
    void memo(char c);
    void indep(dbinlob *ob); // scoate elementul
    void inceplist(); // afiseaza lista de la inceput spre
                      // sfarsit
    void sfarslist(); // afiseaza lista de la sfarsit spre
                      // inceput

    dbinlob *gaseste(char c); // returneaza pointer spre
                              // elementul cautat

    dbinlob *daincep() { return incep; }
    dbinlob *dasfirs() { return sfirs; }
};
```

Clasa **dllist** întreţine doi pointeri: unul către începutul listei, iar celălalt către sfârşitul ei. După cum puteţi vedea, aceştia sunt pointeri către obiecte de tip **dbinlob**. Aceşti pointeri sunt iniţializaţi cu NULL la crearea listei. Clasa **dllist** admite mai multe operaţii cu listele dublu înlănţuite, incluzând:

- Introducerea unui element în listă
- Scoaterea unui element din listă
- Parcurgerea listei de la început sau de la sfârşit
- Căutarea unui anumit element
- Obţinerea de pointeri către începutul sau către sfârşitul listei

În continuare este examinată fiecare procedură.

Funcţia memo()

Listei i se adaugă informaţii folosind funcţia **memo()**. Ea este introdusă după cum se vede aici:

```
// Adauga urmatoarea intrare.
void dllist::memo(char c)
{
    dbinlob *p;
    p = new dbinlob;
    if(!p) {
          cout << "Eroare de alocare.\n";
          exit(1);
    }

    p->info = c;

    if(incep==NULL) { // primul element din lista
          sfars = incep = p;
    }
    else { // pune la sfarsit
          p->anterior = sfars;
          sfars->urmator = p;
          sfars = p;
    }
}
```

Înainte ca un element să poată fi introdus în listă, trebuie creat de un obiect de tip **dbinlob** care să îl memoreze. Deoarece listele înlănţuite sunt structuri de date alocate dinamic, este normal ca **memo()** să obţină un obiect dinamic, folosind **new**. După ce s-a alocat memorie pentru un obiect de tip **dbinlob**, **memo()** atribuie informaţia transmisă prin **c** membrului **info** al noului obiect şi apoi adaugă obiectul la sfârşitul listei. Reţineţi că pointerii **incep** şi **sfars** sunt actualizaţi în concordanţă cu situaţia. În acest fel, **incep** şi **sfars** vor indica mereu către începutul şi, respectiv, sfârşitul listei.

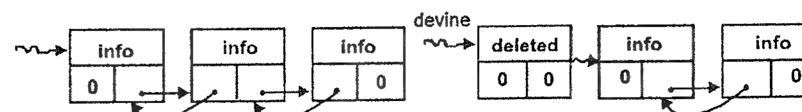
Deoarece obiectele sunt adăugate întotdeauna la sfârşitul listei, lista nu este sortată. Puteţi modifica **memo()**, dacă doriţi, astfel încât să întreţină o listă sortată.

Funcţia indep()

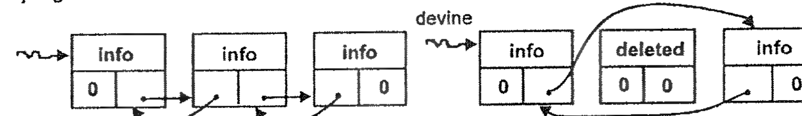
Funcţia **indep()** scoate un obiect din listă. Ea este prezentată aici:


```
/* Indeparteaza un element din lista si reactualizeaza
   pointerii incep si sfars
*/
void dllist::indep(dbinlob *ob)
{
    if(ob->anterior) { // nu este vorba de primul element
          ob->anterior->urmator = ob->urmator;
          if(ob->urmator) // nu este vorba de ultimul element
                ob->urmator->anterior = ob->anterior;
          else // indeparteaza ultimul element
                sfars = ob->anterior; // reactualizeaza
                                      // pointerul sfirs
    }
    else { // indeparteaza primul element
          if(ob->urmator) { // lista nu este goala
                ob->urmator->anterior = NULL;
                incep = ob->urmator;
          }
          else // lista este goala acum
                incep = sfars = NULL;
    }
}
```

Ştergerea primului articol



Ştergerea unui articol din interior



Ştergerea ultimului articol

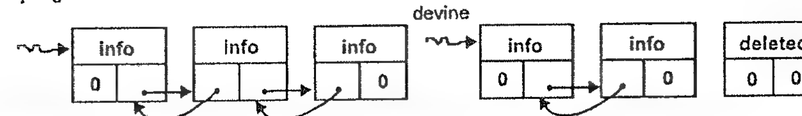


Figura 25-2. *Eliminarea unui obiect dintr-o listă dublu înlănţuită*

Funcția **indep()** scoate din listă obiectul spre care indică parametrul său **ob**. (**ob** trebuie să fie un pointer valid spre un obiect de tip **dbinlob**.) Un obiect care trebuie scos poate să se găsească în trei locuri (vezi **Figura 25-2**). El poate fi primul element, ultimul sau undeva între acestea. Funcția **indep()** tratează toate trei cazurile.

Rețineți că funcția scoate un obiect din listă, dar acesta nu este distrus. El este pur și simplu „desprins din înlănțuire". (Desigur, puteți să îl distrugeți dacă doriți, folosind **delete**.)

Ca și **memo()**, operația efectuată de **indep()** nu depinde de tipul de date care sunt memorate efectiv în listă.

Afişarea listei

Funcțiile **inceplist()** și **sfarslist()** afișează conținutul listei de la început și, respectiv, de la sfârșit. Aceste funcții sunt incluse pentru a vă ilustra cum lucrează clasa **dllist**. Ele oferă un ajutor pentru depanarea programelor.

```
// Parcurge lista de la inceput spre sfarsit.
void dllist::inceplist()
{
    dbinlob *temp;

    temp = incep;
    do {
          cout << temp->info << " ";
          temp = temp->daurmator();
    }     while(temp);
    cout << "\n";
}

// Parcurge lista de la sfarsit catre inceput.
void dllist::sfirsit()
{
    dbinlob *temp;

    temp = sfirs;
    do {
          cout << temp->info << " ";
          temp = temp->daanterior();
    }     while(temp);
    cout << "\n";
}
```


Găsirea unui obiect din listă

Funcția **gaseste()**, prezentată aici, returnează un pointer spre obiectul din listă care conține informațiile ce coincid cu cele specificate ca parametru. Ea va returna **NULL** dacă nu se găsește nici un obiect corespunzător.

```
// Gaseste un obiect pe baza informatiilor sale.
dbinlob *dllist::gaseste(char c)
{
    dbinlob *temp;

    temp = incep;

    while(temp) {
          if(c==temp->info) return temp; // gasit
          temp = temp->daurmator();
    }
    return NULL; // nu este in lista
}
```

Un exemplu de program de listă dublu înlănțuită

Iată clasele complete **dbinlob** și **dllist**, împreună cu o funcție **main()** care ilustrează utilizarea lor:

```
// O clasa negenerica de liste dublu inlantuite.

#include <iostream.h>
#include <string.h>
#include <stdlib.h>

class dbinlob {
public:
    char info; // informatii
    dbinlob *urmator; // pointer spre obiectul urmator
    dbinlob *anterior; // pointer spre obiectul anterior
    dbinlob() {
          info = 0;
          urmator = NULL;
          anterior = NULL;
    };
    dbinlob(char c) {
```

```
        info = c;
        urmator = NULL;
        anterior = NULL;
    }
    dbinlob *daurmator() {return urmator;}
    dbinlob *daanterior() {return anterior;}
    void dainfo(char &c) { c = info;}
    void schimba(char c) { info = c; } // modifica un element

    // Supraincarca << pentru obiecte de tip dbinlob.
    friend ostream &operator<<(ostream &stream, dbinlob o)
    {
        stream << o.info << "\n";
        return stream;
    }

    // Supraincarca << pentru pointeri spre obiecte de tip
    // dbinlob.
    friend ostream &operator<<(ostream &stream, dbinlob *o)
    {
        stream << o->info << "\n";
        return stream;
    }

    // Supraincarca >> pentru referinte dbinlob.
    friend istream &operator>>(istream &stream, dbinlob &o)
    {
        cout << "Introduceti informatiile: ";
        stream >> o.info;
        return stream;
    }
};

class dllist : public dbinlob {
    dbinlob *incep, *sfars;
public:
    dllist() { incep = sfars = NULL; }
    void memo(char c);
    void indep(dbinlob *ob); // scoate elementul
    void inceplist(); // afiseaza lista de la inceput spre
                      // sfarsit
    void sfarslist(); // afiseaza lista de la sfarsit spre
                      // inceput
    dbinlob *gaseste(char c); // returneaza pointer spre
                              // elementul cautat

    dbinlob *daincep() { return incep; }
    dbinlob *dasfirs() { return sfirs; }
};

// Adauga urmatoarea intrare.
void dllist::memo(char c)
{
    dbinlob *p;
    p = new dbinlob;
    if(!p) {
        cout << "Eroare de alocare.\n";
        exit(1);
    }

    p->info = c;

    if(incep==NULL) { // primul element din lista
        sfars = incep = p;
    }
    else { // pune la sfarsit
        p->anterior = p;
        sfars->urmator = p;
        sfars = p;
    }
}

/* Indeparteaza un element din lista si reactualizeaza
   pointerii incep si sfars
*/
void dllist::indep(dbinlob *ob)
{
    if(ob->anterior) { // nu este vorba de primul element
        ob->anterior->urmator = ob->urmator;
        if(ob->urmator) // nu este vorba de ultimul element
            ob->urmator->anterior = ob->anterior;
        else // indeparteaza ultimul element
            sfars = ob->anterior; // reactualizeaza
                                  // pointerul sfars
    }
```

```
   else { // indeparteaza primul element
         if(ob->urmator) { // lista nu este goala
               ob->urmator->anterior = NULL;
               incep = ob->urmator;
         }
         else // lista este goala acum
               incep = sfars = NULL;
   }
}

// Parcurge lista de la inceput spre sfarsit.
void dllist::inceplist()
{
   dbinlob *temp;

   temp = incep;
   do {
         cout << temp->info << " ";
         temp = temp->daurmator();
   }     while(temp);
   cout << "\n";
}

// Parcurge lista de la sfarsit catre inceput.
void dllist:: sfarslist()
{
   dbinlob *temp;

   temp = sfars;
   do {
         cout << temp->info << " ";
         temp = temp->daanterior();
   }     while(temp);
   cout << "\n";
}

// Gaseste un obiect pe baza informatiilor sale.
dbinlob *dllist::gaseste(char c)
{
   dbinlob *temp;

   temp = incep;
```



```
   while(temp) {
         if(c==temp->info) return temp; // gasit
         temp = temp->daurmator();
   }
   return NULL; // nu este in lista
}

main()
{
   dllist lista;
   char c;
   dbinlob *p;

   lista.memo('1');
   lista.memo('2');
   lista.memo('3');

   // foloseste functiile membre pentru a afisa lista
   cout << "Iata lista de la inceput, apoi de la sfarsit.\n";
   lista.inceplist();
   lista.sfarslist();

   cout << endl;

   // parcurge lista "manual"
   cout << "Parcurgere manuala a listei.\n";
   p = lista.daincep();
   while(p) {
         p->dainfo(c);
         cout << c << " ";
         p = p->daurmator(); // da urmatorul
   }

   cout << endl << endl;

   // cauta un element
   cout << "Cauta elementul 2.\n";
   p = lista.gaseste('2');
   if(p) {
         p->dainfo(c);
         cout << "Am gasit: " << c << endl;
   }
```

```
cout << endl;

// scoate un element
p->dainfo(c);
cout << "Scoate elementul " << c << ".\n";
list.indep(p);
cout << Iata lista de la inceput.\n";
lista.inceplist();

cout << endl;

// adauga o alta intrare
cout << "Adauga un element.\n";
lista.memo('4');
cout << "Iata lista de la inceput.\n";
lista.inceplist();

cout << endl;

// modifica informatiile
p = lista.gaseste('1');
if(!p) {
      cout << "Eroare, elementul nu a fost gasit.\n";
      return 1; // eroare
}

p->dainfo(c);
cout << "Modifica " << c << " in 5.\n";
p->modifica('5');
cout << "Iata lista de la inceput, apoi de la sfarsit.\n";
lista.inceplist();
lista.sfarslist();

cout << endl;

// ilustreaza << si >>
cin >> *p;
cout << p;

cout << "Iata lista de la inceput.\n";
lista.inceplist();

cout << endl;
```



```
    // scoate primul element al listei
    cout << "Dupa indepartarea inceputului listei:\n";
    p = lista.daincep();
    lista.indep(p);
    lista.inceplist();

    cout << endl;

    // scoate ultimul element al listei
    cout << "Dupa indepartarea sfirsitului listei:\n";
    p = lista.dasfars();
    lista.indep(p);
    lista.inceplist();

    return 0;
}
```

Iată ieşirea produsă de acest exemplu. (Când programul a solicitat introducerea unei date, s-a scris **X**.)

```
Iata lista de la inceput, apoi de la sfarsit.
1 2 3
3 2 1

Parcurgere manuala a listei.
1 2 3

Cauta elementul 2.
Am gasit: 2

Scoate elementul 2.
Iata lista de la inceput.
1 3

Adauga un element.
Iata lista de la inceput.
1 3 4

Modifica 1 in 5.
Iata lista de la inceput, apoi de la sfarsit.
5 3 4
4 3 5
```

```
Introduceti informatiile: X
Iata lista de la inceput.
X 3 4

Dupa indepartarea inceputului listei:
3 4

Dupa indepartarea sfarsitului listei:
3
```

Crearea unei clase generice de liste dublu înlănţuite

Deşi clasa de liste creată în paragraful precedent este perfect validă, ea poate fi folosită doar pentru liste de caractere, deoarece acesta este tipul de date definite de către **dbinlob**. Dacă aţi vrea să memoraţi alte tipuri de date trebuie să modificaţi specificatorul de tip pentru **info** şi anumite funcţii pentru a le adapta la noul tip de date. Desigur, efectuarea acestor schimbări pentru fiecare tip nou de date este fastidios şi totodată sursă de erori. O soluţie mai bună este să creaţi o clasă generică de liste înlănţuite, folosind un şablon capabil să trateze automat orice tip de date. Exact acest lucru îl face acest paragraf.

Un avantaj de a crea o clasă generică de liste dublu înlănţuite este acela că detaşează mecanismul (adică algoritmul care întreţine lista înlănţuită) de datele memorate efectiv în listă. Astfel, mecanismul poate fi creat o singură dată şi refolosit de oricâte ori.



*NOTĂ: Noţiunile de bază despre crearea şi utilizarea unei clase generice sunt discutate în Capitolul 20. Dacă nu sunteţi familiarizat cu folosirea cuvântului cheie **template** sau cu clasele generice, în general, va trebui să citiţi acest capitol înainte de a încerca să înţelegeţi crearea unei clase generice de liste înlănţuite.*

Versiunea generică a clasei de liste înlănţuite

Primul pas pentru transformarea clasei **dbinlob** şi **dllist** în clase generice este să le declaraţi în şabloane. O dată făcut acest lucru, tipul de date asupra căruia vor opera este transmis ca parametru ori de câte ori este creat un obiect al acestei clase. Iată versiunile generice pentru **dbinlob** şi **dllist**:

```
template <class DataT> class dbinlob {
public:
```

```
DataT info; // informatie
dbinlob<DataT> *urmator; // pointer catre urmatorul obiect
dbinlob<DataT> *anterior; // pointer catre obiectul anterior
dbinlob() {
      info = 0;
      urmator = NULL;
      anterior = NULL
};
dbinlob(DataT c) {
      info = c;
      urmator = NULL
      anterior = NULL;
};

dbinlob<DataT> *daurmator() {return urmator;}
dbinlob<DataT> *daanterior() {return anterior;}
void dainfo(DataT &c) { c = info;}
void modif(DataT c) { info = c; } // modifica un element

// Supraincarca << pentru obiecte de tip dbinlob.
friend ostream &operator<<(ostream &stream,
                           dbinlob<DataT> o)
{
      stream << o.info << "\n";
      return stream;
}

// Supraincarca << pentru pointeri spre obiecte de tip
// dbinlob.
friend ostream &operator<<(ostream &stream,
                           dbinlob<DataT> *o)
{
      stream << o->info << "\n";
      return stream;
}

// Supraincarca >> pentru referinte dbinlob.
friend istream &operator>>(istream &stream,
                           dbinlob<DataT> &0)
{
      cout << "Introduceti informatiile: ";
      stream >> o.info;
      return stream;
```

```
    }
};

template <class DataT> class dllist : public dbinlob<DataT> {
    dbinlob<DataT> *incep, *sfirs;
public:
    dllist() { incep = sfars = NULL; }
    void memo(DataT c);
    void indep(dbinlob<DataT> *ob); // scoate elementul
    void inceplist(); // afiseaza lista de la inceput spre
                      // sfarsit
    void sfarslist(); // afiseaza lista de la sfarsit spre
                      // inceput
    dbinlob<DataT> *gaseste(DataT c); // returneaza pointer
                                       // spre elementul cautat
    dbinlob<DataT> *daincep() { return incep; }
    dbinlob<DataT> *dasfars() { return sfars; }
};
```

După cum puteți vedea, tipul de date generic este numit **DataT**. El este folosit ca un specificator de tip pentru toate referințele la datele memorate în **dbinlob**. Când este creat un obiect, acest tip este înlocuit cu tipul efectiv specificat.

De exemplu, pentru a crea o listă înlănțuită numită **listamea**, care poate memora valori de tip **unsigned long**, veți folosi această declarare:

```
dllist<unsigned long> listamea;
```

Aceasta obține un exemplar al versiunii pentru **dllist** care este capabil să memoreze întregi lungi fără semn. Fiți atenți în declarațiile pentru **dbinlob** și **dllist** la modul în care este tratat tipul generic **DataT** când **dbinlob** este moștenit de **dllist**. Anume, tipul de date folosit pentru exemplarul pentru **dllist** este pasat, de asemenea, către **dbinlob**. Astfel, în declararea anterioară tipul de date **unsigned long** este pasat clasei **dllist** care îl transmite mai departe clasei **dbinlob**. Aceasta înseamnă că, în situația respectivă, tipul de date memorat de un obiect de tip **dbinlob** va fi **unsigned long**.

Pentru a crea un alt tip de listă, modificați pur și simplu specificarea tipului de date. De exemplu, următoarea instrucțiune creează o listă pentru memorarea pointerilor către variabile de tip caracter:

```
dllist<char *> CarPointList;
```


Clasa generică pentru liste dublu înlănțuite

În continuare este prezentată întreaga clasă generică pentru liste dublu înlănțuite și un exemplu de funcție **main()**. Rețineți felul în care este folosit tipul de date generic în cadrul definirii funcțiilor. După cum puteți vedea, în toate cazurile, datele din listă asupra cărora se operează au fost specificate cu ajutorul tipului generic **DataT**. Natura concretă a datelor nu este rezolvată până când nu se obține un exemplar efectiv de listă în cadrul funcției **main()**.

```
// O clasa generica de liste dublu inlantuite.
#include <iostream.h>

#include <string.h>
#include <stdlib.h>
template <class DataT> class dbinlob {
public:
    DataT info; // informatii
    dbinlob<DataT> *urmator; // pointer spre obiectul urmator
    dbinlob<DataT> *anterior; // pointer spre obiectul anterior
    dbinlob() {
        info = 0;
        urmator = NULL;
        anterior = NULL;
    };
    dbinlob(DataT c) {
        info = c;
        urmator = NULL;
        anterior = NULL;
    }
    dbinlob<DataT> *daurmator() {return urmator;}
    dbinlob<DataT> *daanterior() {return anterior;}
    void dainfo(DataT &c) { c = info;}
    void schimba(DataT c) { info = c; } // modifica un element

    // Supraincarca << pentru obiecte de tip dbinlob.
    friend ostream &operator<<(ostream &stream,
                               dbinlob<DataT> o)
    {
        stream << o.info << "\n";
        return stream;
    }

    // Supraincarca << pentru pointeri spre obiecte de tip
```

```
    // dbinlob.
    friend ostream &operator<<(ostream &stream,
                               dbinlob<DataT> *o)
    {
          stream << o->info << "\n";
          return stream;
    }

    // Supraincarca >> pentru referinte dbinlob.
    friend istream &operator>>(istream &stream,
                               dbinlob<DataT> &o)
    {
          cout << "Introduceti informatiile: ";
          stream >> o.info;
          return stream;
    }
};

template <class DataT> class dllist : public dbinlob<DataT> {
    dbinlob<DataT> *incep, *sfirs;
public:
   dllist() { incep = sfars = NULL; }
   void memo(DataT c);
   void indep(dbinlob<DataT> *ob); // scoate elementul
   void inceplist(); // afiseaza lista de la inceput spre
                     // sfarsit
   void sfarslist(); // afiseaza lista de la sfarsit spre
                     // inceput

   dbinlob<DataT> *gaseste(DataT c); // returneaza pointer
                                     // spre elementul cautat

   dbinlob<DataT> *daincep() { return incep; }
   dbinlob<DataT> *dasfars() { return sfars; }
};

// Adauga urmatoarea intrare.
template <class DataT> void dllist<DataT>::memo(DataT c)
{
    dbinlob<DataT> *p;

    p = new dbinlob<DataT>;
    if(!p) {
```



```
          cout << "Eroare de alocare.\n";
          exit(1);
    }

    p->info = c;

    if(incep==NULL) { // primul element din lista
          sfars = incep = p;
    }
    else { // pune la sfarsit
          p->anterior = p;
          sfars->next = p;
          sfars = p;
    }
}

/* Indeparteaza un element din lista si reactualizeaza
   pointerii incep si sfars.
*/
template <class DataT>
              void dllist<DataT>::indep(dbinlob<DataT> *ob)
{
    if(ob->anterior) { // nu este vorba de primul element
          ob->anterior->urmator = ob->urmator;
          if(ob->urmator) // nu este vorba de ultimul element
                ob->urmator->anterior = ob->anterior;
          else // indeparteaza ultimul element
                sfars = ob->anterior; // reactualizeaza
                                      // pointerul sfirs
    }
    else { // indeparteaza primul element
          if(ob->urmator) { // lista nu este goala
                ob->urmator->anterior = NULL;
                incep = ob->urmator;
          }
          else // lista este goala acum
                incep = sfars = NULL;
    }
}

// Parcurge lista de la inceput spre sfarsit.
template <class DataT> void dllist<DataT>::inceplist()
{
```

```
// adauga o alta intrare
cout << "Adauga un element.\n";
lista.memo('4');
cout << "Iata lista de la inceput.\n";
lista.inceplist();

cout << endl;

// modifica informatiile
p = lista.gaseste('1');
if(!p) {
      cout << "Eroare, elementul nu a fost gasit.\n";
      return -1; // eroare
}

p->dainfo(c);
cout << "Modifica " << c << " in 5.\n";
p->modifica('5');
cout << "Iata lista de la inceput, apoi de la
          sfarsit.\n";
lista.inceplist();
lista.sfarslist();

cout << endl;

// ilustreaza << si >>
cin >> *p;
cout << p;

cout << "Iata lista de la inceput.\n";
lista.inceplist();
cout << endl;

// scoate primul element al listei
cout << "Dupa indepartarea inceputului listei:\n";
p = lista.daincep();
lista.indep(p);
lista.inceplist();

cout << endl;

// scoate ultimul element al listei
cout << "Dupa indepartarea sfarsitului listei:\n";
```



```
    p = lista.dasfirs();
    lista.indep(p);
    lista.inceplist();

    return 0;
}
```

Programul precedent foloseşte lista dublu înlănţuită pentru a memora date de tip caracter şi este echivalent funcţional cu versiunea de program, prezentată mai devreme în acest capitol, care nu este generică. Dar, motivul pentru a face clasele de liste înlănţuite să fie generice a fost să le permitem să fie folosite cu orice tip de date. Pentru a vedea cât de uşor este acest lucru, încercaţi să înlocuiţi funcţia **main()** din programul anterior cu următoarea, care creează o listă dublu înlănţuită de valori de tip **double**.

```
main()
{
    dllist<double> lista; // creeaza o lista in dubla precizie
    double c;
    dbinlob<double> *p;

    lista.memo('1.1');
    lista.memo('2.2');
    lista.memo('3.3');

    // foloseste functiile membre pentru a afisa lista
    cout << "Iata lista de la inceput, apoi de la
              sfarsit.\n";
    list.inceplist();
    list.sfarslist();

    cout << endl;

    // parcurge lista "manual"
    cout << "Parcurgere manuala a listei.\n";
    p = lista.daincep();
    while(p) {
          p->dainfo(c);
          cout << c << " ";
          p = p->daurmator(); // da urmatorul
    }

    cout << endl << endl;
```

```
    dbinlob<DataT> *temp;

    temp = incep;
    do {
          cout << temp->info << " ";
          temp = temp->daurmator();
    }     while(temp);
    cout << "\n";
}

// Parcurge lista de la sfarsit catre inceput.
template <class DataT> void dllist<DataT>::sfarslist()
{
    dbinlob<DataT> *temp;

    temp = sfars;

    do {
          cout << temp->info << " ";
          temp = temp->daanterior();
    }     while(temp);
    cout << "\n";
}

// Gaseste un obiect pe baza informatiilor.
template <class DataT> dbinlob<DataT>
                          *dllist<DataT>::gaseste(DataT c)
{
    dbinlob<DataT> *temp;

    temp = incep;

    while(temp) {
          if(c==temp->info) return temp; // gasit
          temp = temp->daurmator();
    }
    return NULL; // nu este in lista
}

main()
{
    dllist<char> lista;
    char c;
```



```
    dbinlob<char> *p;

    lista.memo('1');
    lista.memo('2');
    lista.memo('3');

    // foloseste functiile membre pentru a afisa lista
    cout << "Iata lista de la inceput, apoi de la
              sfarsit.\n";
    list.inceplist();
    list.sfarslist();

    cout << endl;

    // parcurge lista "manual"
    cout << "Parcurgere manuala a listei.\n";
    p = lista.daincep();
    while(p) {
          p->dainfo(c);
          cout << c << " ";
          p = p->daurmator(); // da urmatorul
    }

    cout << endl << endl;

    // cauta un element
    cout << "Cauta elementul 2.\n";
    p = lista.gaseste('2');
    if(p) {
          p->dainfo(c);
          cout << "Am gasit: " << c << endl;
    }

    cout << endl;
    // scoate un element
    p->dainfo(c);
    cout << "scoate elementul " << c << ".\n";
    lista.indep(p);
    cout << Iata lista de la inceput.\n";
    lista.inceplist();

    cout << endl;
```

```
// cauta un element
cout << "Cauta elementul 2.2.\n";
p = lista.gaseste(2.2);
if(p) {
      p->dainfo(c);
      cout << "Am gasit: " << c << endl;
}

cout << endl;

// scoate un element
p->dainfo(c);
cout << "Indeparteaza elementul " << c << ".\n";
lista.indep(p);
cout << Iata lista de la inceput.\n";
lista.inceplist();

cout << endl;

// adauga o alta intrare
cout << "Adauga un element.\n";
lista.memo(4.4);
cout << "Iata lista de la inceput.\n";
lista.inceplist();

cout << endl;

// modifica informatiile
p = lista.gaseste(1.1);
if(!p) {
      cout << "Eroare, elementul nu a fost gasit.\n";
      return 1; // eroare
}

p->dainfo(c);
cout << "Modifica " << c << " in 5.5.\n";
p->modifica(5.5);
cout << "Iata lista de la inceput, apoi de la
            sfarsit.\n";
lista.inceplist();
lista.sfarslist();
```



```
      cout << endl;

      // ilustreaza << si >>
      cin >> *p;
      cout << p;

      cout << "Iata lista de la inceput.\n";
      lista.inceplist();

      cout << endl;

      // Scoate primul element al listei
      cout << "Dupa indepartarea inceputului listei:\n";
      p = list.daincep();
      lista.indep(p);
      lista.inceplist();

      cout << endl;

      // scoate ultimul element al listei
      cout << "Dupa indepartarea sfarsitului listei:\n";
      p = lista.dasfars();
      lista.indep(p);
      lista.inceplist();

      return 0;
}
```

După ce scrieţi această funcţie **main()** în program, ea va determina următoarea ieşire:

```
Iata lista de la inceput, apoi de la sfarsit.
1.1 2.2 3.3
3.3 2.2 1.1

Parcurgere manuala a listei.
1.1 2.2 3.3

Cauta elementul 2.2
Am gasit: 2.2

Scoate elementul 2.2.
Iata lista de la inceput.
1.1 3.3
```

```
Adauga un element.
Iata lista de la inceput.
1.1 3.3 4.4

Modifica 1.1 in 5.5.
Iata lista de la inceput, apoi de la sfarsit.
5.5 3.3 4.4
4.4 3.3 5.5

Introduceti informatiile: 99.99
Iata lista de la inceput.
99.99 3.3 4.4

Dupa indepartarea inceputului listei:
3.3 4.4

Dupa indepartarea sfarsitului listei:
3.3
```

Ar trebui să încercaţi acum singuri să creaţi liste pentru alte tipuri de date. Amintiţi-vă că pot fi memorate în listă chiar şi tipurile de date compuse, cum ar fi structurile care conţin o adresă poştală.

Alte implementări

Există multe feluri în care poate fi utilizată o clasă de liste înlănţuite. Puteţi să faceţi încercări şi pe cont propriu. Iată câteva idei cu care aţi putea să începeţi.

Listele din acest capitol adaugă, pur şi simplu, obiecte la sfârşitul listei. Acest lucru este acceptabil (ba chiar de dorit) pentru multe aplicaţii. Totuşi, aţi putea modifica **memo()** astfel încât să creeze o listă sortată sau crea o altă versiune care să adauge elemente la începutul listei. De fapt, puteţi să definiţi mai multe versiuni ale funcţiei **memo()** (fiecare cu propriul său nume, care să o explice) ce memorează elemente în diferite feluri. De exemplu, puteţi defini funcţii denumite **MemoSfars()**, **MemoIncep()** şi **MemoSort()**, care să adauge elemente la sfârşit, la început sau, respectiv, într-o anumită ordine.

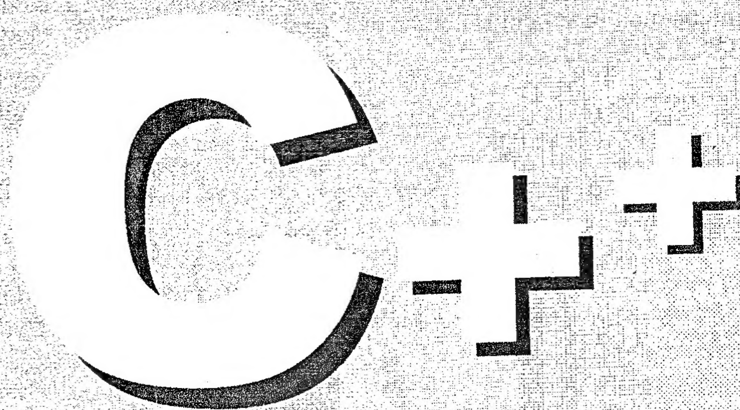
O funcţie pe care poate doriţi să o adăugaţi este numită **dalungime()**. Faceţi-o să returneze numărul de elemente din listă.

Aşa cum s-a menţionat mai devreme, pointerii **urmator** şi **anterior** au fost declaraţi publici intenţionat, pentru a simplifica rutinele pentru listele înlănţuite şi pentru a ilustra complet clasele de liste înlănţuite. Puteţi însă stabili ca aceşti pointeri să fie particulari, protejându-i astfel împotriva utilizărilor greşite.

Un ultim lucru: chiar dacă listele înlănţuite prezentate în acest capitol memorează caractere şi numere în virgulă mobilă, amintiţi-vă că poate fi memorat orice tip de date.

Anexa A

Bibliotecile de clase standard propuse



Comitetul de standardizare C++ este în procesul de definitivare a setului de biblioteci de clase standard. Acum, în momentul scrierii acestei cărţi, ele sunt încă în lucru şi nu sunt acceptate complet de nici un compilator de C++ disponibil. De aceea, nu este oportun să discutăm bibliotecile de funcţii în această ediţie a cărţii. (Excepţie face doar biblioteca de I/O, care este introdusă curent în toate compilatoarele şi care este discutată în amănunţime în **Partea a doua.**) Totuşi, deoarece compilatoarele viitoare le vor conţine, este important pentru dvs. să cunoaşteţi ce veţi avea la dispoziţie. Iată, deci, lista cu bibliotecile care sunt definite curent de către standardul propus ANSI C++:

- Sprijin pentru limbaj
- Diagnostice
- Utilizare generală
- Şiruri
- Localizări
- Recipiente
- Iteratori
- Algoritmi
- Numerice
- Intrări/Ieşiri (I/O)

Chiar dacă standardul ANSI C++ este încă în stadiu de dezvoltare, ar fi bine să verificaţi manualul compilatorului dvs. pentru a vedea care dintre aceste biblioteci de clase sunt admise.



REŢINEŢI: Bibliotecile de clase se adaugă bibliotecii de funcţii standard, care este inclusă în toate compilatoarele de C++.